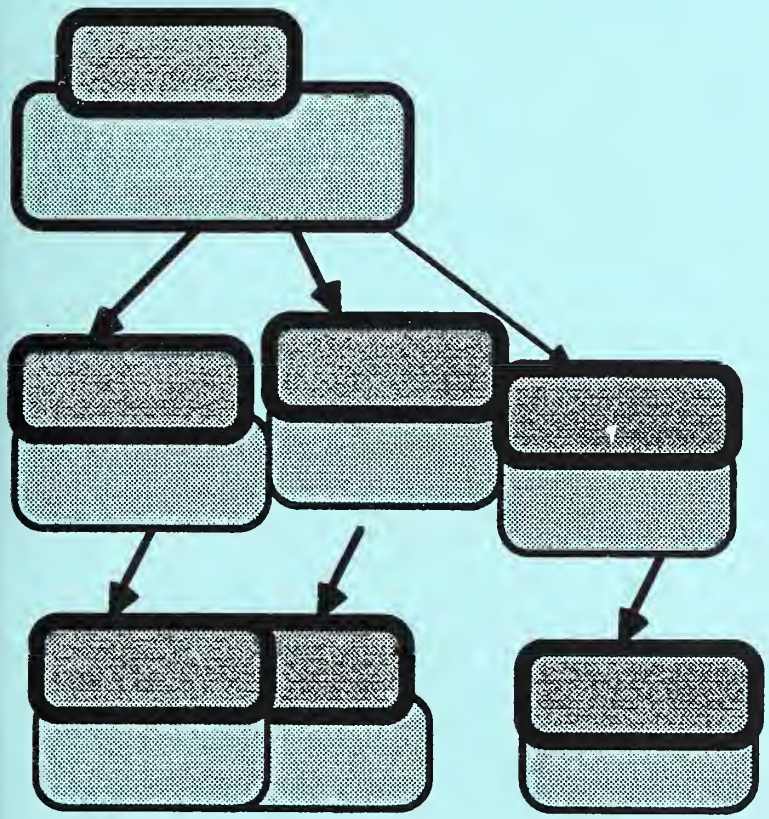
NISTIR 4488

PROCEEDINGS OF THE OBJECT-ORIENTED DATABASE TASK GROUP WORKSHOP TUESDAY, OCTOBER 23, 1990 CHATEAU LAURIER HOTEL OTTAWA, CANADA



Elizabeth N. Fong
Editor

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Gaithersburg, MD 20899

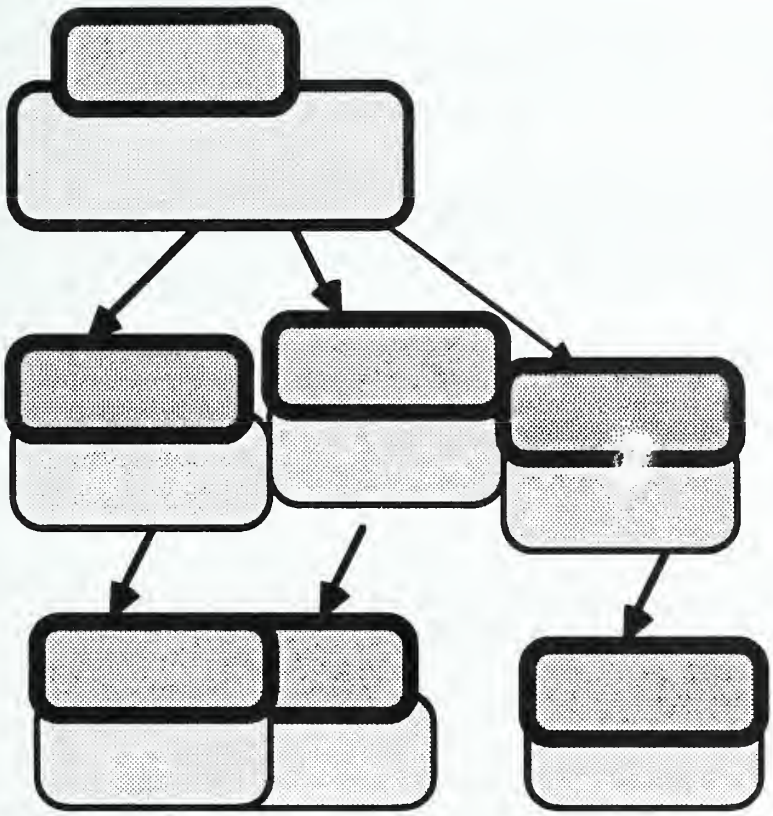
U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4488

PROCEEDINGS OF THE OBJECT-ORIENTED DATABASE TASK GROUP WORKSHOP TUESDAY, OCTOBER 23, 1990 CHATEAU LAURIER HOTEL OTTAWA, CANADA



Elizabeth N. Fong
Editor

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
National Computer Systems Laboratory
Gaithersburg, MD 20899

January 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

PREFACE

This volume is a collection of papers from a workshop held by the Object-Oriented Databases Task Group (OODBTG) of the Database Systems Study Group (DBSSG). The DBSSG is one of the advisory groups to the Accredited Standards Committee X3 (ASC/X3), Standards Planning and Requirements Committee (SPARC), operating under the procedures of the American National Standards Institute (ANSI). The OODBTG was established in January 1989. Consistent with usual practice when confronted with a complex subject, DBSSG charged the OODBTG to investigate the subject of Object Databases with the objective of determining which, if any, aspects of such systems are, at present, suitable candidates for the development of standards.

This workshop is one of several activites sponsored by OODBTG, which will contribute to the Final Technical Report of OODBTG. The Final Report is scheduled to be delivered to DBSSG in 1991 and will include recommendations as to which standards X3 should attempt to develop in the area of OODB. A similar workshop was held in May 1990 in Atlantic City, New Jersey. These two workshops will provide data points from which recommendations in the Final Report can be developed.

Other activities currently under way by OODBTG include development of a Reference Model containing informal definitions and requirements for an Object Database Management System, and a Survey of OODB Systems

The papers reprinted in this volume represent the opinions of the individual authors. These papers are neither approved standards nor recommendations of OODBTG. Neither OODBTG nor the Program Committee have made any judgements as to whether any topic of any paper complies with the Reference Model currently under development by OODBTG.

For further information about OODBTG, please contact:

Elizabeth Fong
National Institute of Standards and Technology
Building 225, Room A266
Gaithersburg, MD 20899

Finally, I would like to thank the other members of the Program Committee for their efforts in reviewing the papers of this Workshop.

Allen Otis
Portland, Oregon
15 October 1990

ABSTRACT

This report constitutes the proceedings of a one-day workshop on standardization of object database systems held at the Chateau Laurier Hotel, Ottawa, Canada, on October 23, 1990. The workshop was sponsored by the Object-Oriented Database Task Group (OODBTG) of the ASC/X3/SPARC Database Systems Study Group (DBSSG).

This workshop, held on the third day of the Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), was the second attempt to solicit public input to identify what aspects of object database systems may be candidates for consensus that can lead to standards. The first companion workshop was held on May 22, 1990, in Atlantic City, New Jersey, coincident with the International Conference on Management of Data (SIGMOD).

The workshop goals focused on concrete proposals for languages or module interfaces, exchange mechanisms, abstract specifications, common libraries, or benchmarks. The workshop announcement also solicited papers on relationships of object database systems capabilities to existing standards, including assertions that question the wisdom of standardization.

This proceedings consist of 13 position papers covering various aspects where standardization on object database systems may be possible.

Key words: Database; database management system; DBMS; data model; object-oriented; OODB; programming languages; standards.

DISCLAIMER

The views expressed in this report are those of the authors, and do not necessarily reflect the views of the National Institute of Standards and Technology (NIST) or any of its staff. The specific vendors and commercial products identified in this report do not imply recommendation or endorsement by the NIST.

The report has not been subject to policy review or direction by the NIST, nor by Accredited Standards Committee X3 Information Processing Systems, Standards Planning and Requirements Committee (SPARC).

Accredited Standards Committee, X3 INFORMATION PROCESSING SYSTEMS,
SPARC/DBSSG/OODBTG

OODBTG Workshop
Tuesday, October 23, 1990
Chateau Laurier Hotel, Ottawa, Canada

CONTENTS

Aspects of Object System Standardization

Roger Osborn, Michael Maddison, Dennis Layton

Concurrent Computer Corporation
Slough, England

1 Introduction

An object system provides a new and uniform single architectural paradigm, for application systems, that is different from the traditional distinct paradigms of application programs and database. Application and database are merged so that there is no clear separation between them.

Addressing standardization of Object Oriented Databases from a traditional database perspective could cause incorrect balance and lead to unrealistic or irrelevant standards. Instead a total system perspective, that addresses the standardization of object systems as a whole, should be taken.

In this position paper, the benefits and scope of object systems and their standardization are discussed. The fundamental properties of objects and object systems are described. The aspects of object systems that must be considered for standardization, if interoperability and portability are to be promoted, are listed.

2 Object System Standardization Benefits

Object orientation provides a means whereby an application system can be structured so that its organization corresponds closely to the real world or abstract system that it is supporting and thus capture much of its semantics. This has potential benefits in reliability, security, performance, fault tolerance and in the facilitation of the continuous development of an application to match the evolution of its supported system.

Although, in object systems so far produced, all of the above benefits may not have been achieved, standardization should encourage the production of object systems where they are realized. Standardization should also contribute its own benefits of interoperability, portability and approved common and consistent functionality.

3 Object System Scope

The scope of an object system is broad. The traditional two separate models for application programs and database management are replaced by a single model for both programming and managing data. Thus the scope covers all aspects of information processing and management. These include: languages, processing models, distribution, concurrency, persistence, access, consistency, security, user interface, development and maintenance. Object orientation is a general purpose and unifying technology. It is not restricted to a few application areas. With a complete object system there are few limitations on the types of application system that can be supported. Data processing, transaction processing, real-time, fault tolerant, knowledge based, artificial intelligence, etc., applications should all be implementable in an object system and be able to partake of the benefits of such a system.

Standardization should address itself to this broad scope, and establish a base framework for a series of related standards, covering different aspects of object systems, but based upon a common fundamental object model or family of models. This standardization process should not be reluctant to take the lead in mapping out any less understood or established aspects.

4 Object and Object System Fundamentals

4.1 General Features

An object oriented application system consists of objects. An object encapsulates information and behaviours. It supports a number of functions. Processing takes place when objects execute in either direct or indirect response to the invocation of their functions. An object system supports an invocation service.

Objects can invoke each other. An object can be concurrently executing in response to multiple invocations.

An object exists within a context. It exists until either it is deleted or the context terminates. The context can be quite persistent, such as the life of the object system, or it can be transient, such as: a session of the object

system, a user session with the object system, a transaction etc. An object's persistence or transcience can be independent from its other properties. For objects that require persistence, the object system supports an object store.

An object can participate in a transaction such that any changes to it are subject to the commitment of the transaction. An object system contains a transaction manager which supports transaction control functions and co-operates with other systems' transaction managers in the commitment of distributed transactions. An object's participation in a transaction is orthogonal to its degree of persistence.

Only authorised invocations are permitted to security controlled objects. Authorization is either via possession of a capability for the invocation, or via direct or indirect matching of an authenticated invoker against an authorized invoker list.

There is a strong separation between the external interface or view of an object and its internal implementation. The invoker of an object is only concerned with its external interface and has neither awareness of, nor access to, its internal implementation. Thus, for instance, standardization for interoperability is not concerned with the internal implementation of objects.

4.2 External Object Interface

Externally an object has an identity and a type. The identity is unique within some scope that can be either the complete object system or a subset of it.

An object's type defines the functions that it supports. Thus all objects of the same type support the same set of functions. Types form an inheritance lattice where an object of a given type is a specialization of objects of its supertypes. It supports all of the functions that they support plus any additional ones specified with the type. Thus to an invoker an object can be treated as though its type was that of one of its supertypes.

A type is an object in its own right. It supports functions that provide information on its definition, such as descriptions of the functions that are supported by objects of the type. These descriptions include definitions of the syntax of function invocations, that is input and output parameters and their types (integer, real, string, array, etc, or typed object references). Descriptions could also include formal specifications of the functions.

4.3 Internal Object Implementation

An object's implementation is independent of its type and there can be many different implementations for the same type. There are many different possible implementation models. The one, that is now described, is a class based system. In such a system an object's class defines its implementation, that is its instance variables, which represent its state, and the code bodies (or methods), which are executed when its functions are invoked. Objects of the same class share the same methods and have the same instance variable structure. Inheritance between classes is possible, so that a class inherits the instance variable structure and methods from its super classes. The inherited structure can be extended to incorporate locations for new instance variables. New methods are specified as necessary for the corresponding type. An inherited method can be replaced by a new method thus giving a new implementation for a function. Inherited instance variables cannot be directly accessed by new methods as this could lead to unpredictable changes to the behaviour of inherited methods.

A class is an object in its own right and it supports a function 'create instance.' Invocation of this function causes a new object, which is an instance of the class, to be created.

5 Standardization Aspects

5.1 Interoperability

For standardization to promote interoperability, so that an object in one object system can be invoked from another object system or a non-object system, either on the same or a different machine, there are many aspects to be considered. These mainly concern the external interface to objects and an object system and are independent of the internal implementation of an object.

- invocation

Extensions to programming languages to support object invocation statements. Conversions of invocation parameters from one language or system to another. Use of a standard notation such as ASN.1 in such conversions. Invocation protocols including the communication of session information such as user or invoker identity, transaction information. Exception handling.

- identity

The format for object identifiers. Name servers for mapping names to object identifiers.

- transactions

Two or three phase commit protocols to be followed by transaction managers in order to commit a multi-system transaction. Use of standard protocols.

- security

Invoker authentication. Capability or authorised invoker control list handling. Invocation logging.

- type

Standardized type objects to enable function syntax to be retrieved and used either for type checking prior to object invocation or for object browsing.

- subtyping

Facilitates flexibility between invoker and invokee. An invoker need only know the type of an object to the level of specialization that interests it.

- standard types

Standard types for particular application domains. For objects of these types invokers can assume type information.

There are degrees of interoperability varying from the ability to invoke a few static objects to a very dynamic but well controlled interaction between changing groups of objects. Depending upon the degree or quality of interoperability required, different levels of support for the above aspects are necessary.

5.2 Portability

To promote portability so that types and object implementations can be imported into one object system from another, standardization must address the internal and external specifications of objects. For a class based implementation system the ability to import methods, instance variable structures and object classes is required. Aspects, that have to be considered, concern both the importation mechanisms and the assumptions, that method code must make about the environment in which it is to run.

- type

The types of type class and other meta-objects required for type creation. The type inheritance policy.

☐ class

The types of class class and other meta-objects required for class and object creation. The class inheritance policy. The means of invoking superceded superclass methods.

☐ object dependencies

Support for invocation by imported methods of objects of other types either concurrently imported, newly created or pre-existing. Standard type assumptions.

☐ languages

Extensions to programming languages to enable methods to access instance variables and to invoke other objects.

☐ process model

Synchronous invocation assumption. Mechanism for initiating asynchronous activity. Concurrency control mechanism. Exception handling mechanism. Process functions.

☐ object reference

Functions for assignment and comparison.

☐ persistence

Mechanism for saving updated state to persistent storage.

☐ transactions

Transaction model including nested and asynchronous transaction support. Nested transactions enable objects to control their own recovery and consistency independently from the transaction context of their invocation. Locking model. Commitment assumptions, either transparent to objects or object supported commit functions. Transaction manager functions.

☐ security

Functions for granting, preserving and other manipulations of capabilities. Alternative functions for granting and revoking rights.

☐ class libraries

Mechanisms for their importation. Importation with corresponding types. Matching with existing or standard types.

Several standards could result from consideration of the above aspects. They all need to be considered if portability is to be fully realized.

6 Conclusions

Object orientation, if fully supported by an object system provides a simpler, more powerful and complete paradigm for computation than the traditional application program and database paradigms. It is applicable to most application domains. Because of this broad applicability, a framework for standardization is necessary, and many aspects need to be considered. We have listed those aspects, that we believe should be considered, if interoperability and portability are to be promoted.

A Strawman Reference Model for an Application Program Interface to an Object-Oriented Database

Edward Perez
Texas Instruments Incorporated *

Abstract

This position paper provides a strawman reference model which can be used to compare and reason about an application program's interface to an Object-Oriented Data Base system (OODBs), with an emphasis on object manipulation from an object-oriented programming language bindings. [1] We begin with an introduction of OODBs and describe the broad domain for an OODB reference model as well as the specific sub-domain we will discuss. Next, we describe some goals of the reference model for the specific sub-domain. A *descriptive* reference model is then described as consisting of a collection of design choices that can be used for comparing existing and future application program interfaces to OODBs. Finally, based on the descriptive reference model, we define a *functional* reference model that provides a much more precise description of the interface provided by one such OODB, Zeitgeist, being developed at Texas Instruments.

1 Introduction

In the past few years, new applications have been developed which have data modeling needs that are much more complex, both in content and interobject relationships. To assist in the development of these applications, new object-oriented languages have been developed to provide an application developer the ability to create and manipulate complex data inherent in these applications. Several research prototypes and commercial object-oriented databases systems (OODBs) have been developed in the past few years (including Iris [1, 2, 3], Orion [4, 5], E [6], Postgres [7, 8, 9], ODE [10], GemStone [11], Ontos [12], OBJECT-Base [13], ObjectStore [14] and Zeitgeist [15]) to provide long term storage of the data created by these applications.

Several papers [17, 18, 19, 20] have described the main features and characteristics that a system must have in order to qualify as an object-oriented and/or third generation database. Within the broad domain of OODBs, we see the following sub-domains.

1. **Language Independent Data Model**

   OODBs in this sub-domain (e.g., Postgres and Iris) provide a new database language to define types and classes as well as to manipulate and query instances of the predefined classes (the languages have a heritage from the relational database languages). The language is also to be considered independent of application programming languages. Persistence is implicit for each class defined.

2. **Language Dependent Data Model**

   OODBs in this sub-domain provide similar features but also allow for explicit navigation between objects.

   (a) **New language**

   OODBs in this sub-domain (e.g., Trellis/Owl) develop a new language and add persistence to classes in the language.

   (b) **Existing language**

   OODBs in this sub-domain extend an existing language and add persistence to classes in the language.

   i. **Dual type system**

   OODBs in this sub-domain (e.g., Orion, Gemstone, E) do not separate the concept of persistence from type, requiring the application developer to define two classes/types, one for transient and one for persistent usage.

   ii. **Single type system**

   OODBs in this sub-domain (e.g., ObjectBase, ODE, Ontos, VERSANT

*Information Technologies Laboratory, Technical Report 90-07-03, Computer Science Center, Texas Instruments Incorporated, P.O Box 655474, MS 238, Dallas, Texas 75265. Email: perez@csc.ti.com Telephone: (214) 995-0698

[1] Other aspects of an OODB API are covered in [21, 22, 23].

Manager, Zeitgeist) separate the concept of persistence from type.

In general, an API to an OODB includes the ability to 1) allow the application developer to define types and classes to the system, 2) allow the application developer to manipulate objects (also called instances) of the defined classes, and 3) allow the application developer and/or end-user to query the objects within the context of an application program or a stand-alone query system. In this paper, we will concentrate on defining an API reference model for those OODBs in sub-domain 2(b).ii (an OODB based on an existing object-oriented language and data model with a single type system) and, more specifically, concentrate on an API for manipulating objects. Since we are concentrating on this sub-domain, issues related to data model definitions are best left to a discussion of the specific programming languages and their extensions (although, there is a short discussion in Appendix A), while issues related to queries, transactions, and change management are best left to other position papers, such as [21, 22, 23]. Finally, although many of the concepts presented here are applicable to the other OODB categories, other position papers are needed to fully develop the overall OODB API reference model.

Section 2 discusses some of the goals of the chosen sub-domain and API. Section 3 presents a *descriptive reference model* that provides a framework that can be helpful for future standardization efforts in the area of APIs to OODBs. The reference model describes a design space of characteristics and defines the criteria and features that serve as a basis for comparing different existing and future APIs. Section 4 describes a *functional reference model* using the API of one specific OODB, Zeitgeist. Section 5 presents a comparison between a representative sample of OODB APIs using the established reference model. Finally, we present our conclusions in Section 6.

2 Goals

The traditional goals of database systems have been to manage information in a persistent and recoverable storage medium that can be shared in a controlled manner by multiple users and/or applications. For the most part, the API to these databases has been accomplished by embedding a data manipulation language in a conventional programming language or by using some form of an interactive query facility.

While the usefulness and productivity improvements gained from the use of query facilities has been substantial, the embedding of a data manipulation or query language in programming languages has constrained application developers by forcing them to conceptualize, design, and implement their application using two or more data models and languages.

Thus, we believe that an OODB should help to improve the productivity of application developers in at least the following ways. First (Goal 1), the **impedance or mismatch** between the data models of the programming language and the database can be virtually eliminated by using one data model (and language) to manipulate the objects instead of translating between the two models. Second (Goal 2), productivity can also be increased by reducing the **number of explicitly coded interactions** between the application program and the database, especially for retrieval of objects from the database. Finally (Goal 3), the introduction of an OODB to an existing application program (whether or not it is using a database system) should minimize the **additions to or modifications of the application program's code** (whether modified by the application developer or the OODB system). As we describe the reference model, we will refer back to these goals.

3 Descriptive Reference Model

In this section, we present a *descriptive reference model* for an application program interface to an OODB.

3.1 Common Features

In this section, we describes certain *common features* that we believe must be included in any API to an OODB. They can be used as a starting point for developing a consensus that can lead to standards in the area of an OODB API. We separate these features into those that pertain to the database system and those that pertain to the individual objects manipulated by the application program and managed by the database system. Unless otherwise stated, we will use the term "system" to refer to the database system.

3.1.1 System Interface

Regardless of the database system used, certain interfaces must be supplied to allow the application program to define the boundaries of database and transaction execution.

System Startup and Shutdown

Prior to using any of the interfaces defined in the API, the application program must indicate its intent to begin interacting with the system to allow the latter to perform whatever actions are necessary to make itself ready for subsequent requests from the application program. Once the application program has completed its interaction with the system, it must indicate that no further interactions with the system will occur to allow the latter to perform whatever actions are necessary to orderly shut down or terminate itself. One of the design choices is whether the startup and shutdown must be performed explicitly by the application developer or implicitly by the system. Although these actions are not performed frequently during the execution of an application program, if they can be performed implicitly (perhaps by exploiting the semantics of programming language variable allocation and deallocation), this will contribute to satisfying Goal 2.

Beginning and Ending Transactions

Although certain interactions with the system do not affect the state of the database (e.g., obtaining the release version, setting any performance controls, etc.), interactions involving class definitions or objects must take place within the boundaries of a transaction to insure the integrity of the state of the database. The application program must indicate the beginning of the transaction. As a result of this indication, the system may take whatever actions are necessary to prepare for subsequent creation and/or retrieval of objects, requests for object locks, etc. When the application program has completed an appropriate set of database and programming operations involving newly-allocated and/or retrieved objects, it must indicate the end of the transaction and specify the disposition of the persistent objects manipulated during the transaction. An application program may choose to commit the work, in which case the persistent objects are saved, altering the state of the database, or abort the work, in which case the state of the database will not be altered. In addition to processing the end of the transaction, the system may perform whatever actions are necessary to properly end the transaction. Although the boundaries of a transaction must be explicitly indicated (How many run-time systems can automatically infer the intent of an application programmer?), another design choice is whether the application developer uses the OODB API directly or whether programming language constructs should be defined which would implicitly use the specific OODB interfaces.

3.1.2 Individual Object Interface

In this section, we discuss the interfaces to allocate persistent objects, manipulate their database status via **references** (also called handles), and retrieve them from the database. Manipulation of the actual objects (via their references) is performed using the methods defined for the object's class and will not be discussed in this paper.

Allocating and Deallocating Objects

The application programmer decides when an object comes into existence and when it is no longer needed. Some of the objects allocated and used within an application program will exist only during all or part of an execution of an application program (**transient**) while others will exist during an execution of an application program as well after it has terminated its execution (**persistent**). However, some transient objects can **never** become persistent (*they are fully transient*), while others **may** become persistent (*they are potentially persistent*). Therefore, in this paper, a class whose objects will never become persistent will be called a **transient class** and its objects **transient objects**. A class whose objects might become persistent will be called a **persistent class** and its objects **persistent objects**. Throughout this paper, we will use the word "object" and "instance" interchangeably, adding the adjectives "persistent" and "transient" as needed.

The design choice concerns the programming language construct used to allocate an object: should the allocation be dependent upon the object's persistence? That is, should the construct used to allocate a transient object be the same or different as the one used to allocate a persistent object? Since, in sub-domain 2(b).ii, persistence is orthogonal to the object's class (type), using the same construct to allocate both kinds of objects helps to satisfy Goal 1 as well as Goal 3. When coding the actual allocation of the object, the application programmer does not need to ask questions such as "Will this object ever be saved to the database?" ("It should not matter at this point in time.") or "Which language con-

struct do I use to allocate this object?" ("The construct defined by the language, of course."). In addition, this orthogonality of persistence would also allow a class library to be developed with persistence comprehended, leaving to the users of the library the decision of whether or not to make instances of the library classes persistent. Similarly, the construct used to deallocate transient and persistent objects should be the same. The issue of physical deletion of a persistent object is in another design space and will not be considered in this paper.

Assigning and Comparing References to Persistent Objects

After the application programmer has allocated a persistent object, (s)he must be able to assign a reference to it as well as compare references to persistent objects. Analogously to allocating persistent objects and retrieving persistent objects (see below), the design choice concerns the language construct used to perform these assignments and comparisons of references to transient and persistent objects: should they be the same or different. If the same construct can be used to manipulate a reference to a persistent or transient object, we can further satisfy Goal 1. In addition, use of the same construct helps to satisfy Goal 3 by eliminating the need to add additional code to accomplish these tasks. At the API level, we will only require that the ability to assign a reference to a persistent object, assign one reference to another, and to compare the values of two references (i.e., virtual memory address of the objects) be supported. Assigning or comparing beyond the values of the references is a design choice in the design space of the data model and should be made by the data model, language and/or application developers and not the OODB system.

Indicating Persistence of Objects

As stated before, we distinguished between classes whose instances would never be saved to the database and classes whose instances might be saved to the database. Clearly, indicating persistence can only apply to instances of the latter classes and must be made prior to the end of the current transaction. The design choice concerns when that indication of persistent is made: implicitly when the object is allocated or explicitly after the object has been allocated and before the end of the current transaction. We believe that indicating persistence of an object should be orthogonal not only to its type but to its allocation. This choice provides the application programmer with more flexibility at the expense of an additional interaction with the system. The application program performs only what is necessary when it is necessary; a more economical and "only pay for what you use" philosophy.

Indicating Modification to Persistent Objects

When an persistent object is allocated or retrieved, the system has to determine whether or not the object should be saved to the database if the current transaction is committed. This determination could be made by 1) relying on the object's existence in virtual memory, 2) detecting modifications by having the compiler add code to flag modifications to the object, 3) detecting any writing to the object, or 4) having the application program explicitly inform the system that the object has been modified. Option 1 is the easy way out but probably results in saving more objects than are necessary (not every object created or retrieved is modified). Option 2 involves additions to every compiler that can compile code which interfaces with the system and determining which modifications imply an actual change to the object. Option 3 requires special hardware or software on every system where the application program will execute. Option 4, however, leaves the choice to the application developer and follows the "only pay for what you use" philosophy. With options 2-4, we now have two conditions a persistent object must satisfy before it can be saved to the database: it must have been indicated as being persistent **and** as having been modified. With option 4, we have simplicity and flexibility at the expense of an additional interaction with the system.

Retrieving Persistent Objects

After an object has been saved to the database as the result of committing a transaction, it can be retrieved at some point in the future by the same or another application program. Clearly, the API must provide an interface to allow explicit retrieval of objects. Here, the design choice concerns whether or not the application program must explicitly retrieve every object it needs. If, when an application program dereferences a reference to a persistent object (i.e., evaluates the value of the reference to obtain the address of the persistent object), the system can implicitly retrieve the object on behalf of the application program, then we can further satisfy Goal 2. We call this implicit retrieval **object faulting**. Although this incurs a small run-time overhead (to test and see if the object is currently in virtual, or

primary, memory), we believe that it is much more preferable than *requiring* the application to continually perform this test, which, if omitted or forgotten by the application developer, would result in an unexpected (from the developer's point of view) software or hardware exception or interrupt. Dereferencing an invalid reference to a transient object should not be detected nor handled by the system, other than invoking any necessary recovery actions caused by exceptions or interrupts.

In the above discussion, the persistent object was retrieved into the application program's workspace and manipulated there. If, however, the system can dynamically invoke the methods defined for an object's class (i.e., the system is **active**), then the application developer could ask the system to invoke the method in the system's workspace, thereby reducing the memory requirements of the application's workspace.

3.2 Additional Features

In this section, we describe *additional features* of an API. These features are supported by only a few of the existing systems and require further research and experimentation before an acceptable general consensus regarding each feature can be achieved.

3.2.1 System Interface

Extended Transaction Models

Many applications have transactional requirements that are not fully satisfied by the standard single-level transaction provided by many existing as well as object-oriented database systems. Some applications require transactions that can be nested to allow application developers the freedom to define transaction boundaries without having to consider previously defined transaction boundaries (which may be encapsulated or nested), while other applications require transaction boundaries that are "nested" but commit their modifications to the database independent of the higher level transaction (e.g., a real time manufacturing system). Other applications, such as CAD, CAE, CASE, etc., require transactions to allow a group of individuals to cooperatively share the results of their intermediate transactions prior to making that final commit. Although much research has been performed in this area, we believe that more work is needed to delineate the various categories of transactions and estimate their expected use and impact before standards can be formulated. See [23] for more discussion of these issues.

Grouping Objects

Past work on locality of reference indicates that the application developer should be able to specify some sort of grouping, clustering, or co-locating of objects to improve retrieval performance and, perhaps, object migration. Some of the design choices for this feature include whether the grouping is logical and/or physical, whether the grouping can be defined at object model definition-, compile-, and/or run-time, whether the grouping can be altered for existing and/or newly created objects, and what can be stored in the cluster: objects from just one class or from multiple classes. As with previous API features, these features should be separately available to provide the application developer the flexibility in choosing what is needed for the specific application and to only "pay for what is being used".

3.2.2 Individual Object Interface

Determining and Forcing Memory Residency

Since this model of an API provides several interfaces to manipulate a reference to a persistent object (assign, compare, etc.), we could easily add the ability to allow the application program to determine if a persistent object is or is not currently in memory. Although the object faulting mechanism described earlier performs this function implicitly for the application program, this interface would be provided for completeness, flexibility, and performance. Again, if the application program wants "to pay" for this ability, it can do so separately and at a minimum cost.

Retrieving Previous Versions of Objects

Many OODBs can store more than one version of an object in the database (e.g., [4, 9, 10, 15]). The application program should be able to retrieve a previous version *as easily as* retrieving the most recent version. Assuming that version retrieval is included in the API [2], this can be done by supplying an additional argument when retrieving the object. To help satisfy Goal 3, we should not require a completely different language construct or retrieval

[2] Although retrieval of object versions may be a higher level task [22], eventually some module needs to retrieve a specific object version.

interface to accomplish this task. When the application program specifies the version to be retrieved, can it specify the version "most recent relative to time x" and let the system determine which version meets that specification, does the specification have to be exact using a time stamp or version number, or can the system support both types of specification? Another design choice involves determining whether or not this version specification should be propagated when object faulting is involved: do we retrieve the most recent version of the faulted object or the most recent version relative to the version specified in the referencing object? As versions imply change management, see [22] for more discussion of these issues.

Locking Objects

Since one of the main goals of a database management system is to provide controlled sharing of a base of objects, we must provide the application program the ability to lock an object for **read** or **write** access. This access could be specified as an option when retrieving the object (with the default being predefined by the system or, possibly, the application program). For additional flexibility, the system could provide an interface to upgrade the existing lock on an object after it has be retrieved. For even further flexibility, the system could allow the application program to downgrade an existing lock, at least as far as the application is concerned (due to problems with various two-phase locking protocols). Finally, certain application programs would like to have just a snapshot of the current state of an object regardless of whether it is being updated by another application program. Thus, a **read-only-don't-block** type of lock could also be provided at a very minimal cost to the application program, the system, and other concurrent application programs. The system must also detail what options the application program has when a lock request conflicts. Possible options are waiting for the lock, receiving an error indicating a conflict, or asking for a notification when the object is no longer locked.

Naming Objects

Retrieval of objects is accomplished by specifying a unique identifier for the object, often called an OID or UID. In most OODBs, this identifier is an integer number (32, 64, or 96 bits), not meant to be remembered by the person executing the application program. Thus, there is a need to alternatively identify objects which the application developer can use as needed. The naming of the object should be orthogonal to its allocation and should not require any special declaration when the type is made know to the system or the object is allocated or retrieved. Two alternatives include choosing one or more attributes of an object as a key (as is done in the relational model) or associating a name with an object. The former is easily accomplished and probably more intuitive but requires the maintenance of the appropriate system table(s) whenever an object's key is modified, which contradicts one notion of object identity, that of immutability [24, 25]. Associating a name with an object provides the application program the flexibility in choosing whether or not to specify a name and manage the naming if it chooses to do so. In addition, the name for the object can be easily changed without affecting the object's identity (i.e., its OID) or its value. Another design choice is whether or not the name-to-object mapping should be unique: can one object have multiple names? The flexibility afforded by the file pathname "link" in various files system supports adding this feature to the API.

4 Functional Reference Model

Now that we have presented a descriptive reference model in Section 3 and discussed several features of an API to an OODB, we now present a more detailed description of the common features of the reference model for the Zeitgeist OODB [16]. Since a consensus on the additional features is farther off, we will omit a detailed description of them for future position papers.

In keeping with the object-oriented style, we present the interface in terms of methods and operators associated with two basic classes: the system (Zeitgeist) and references to persistent objects (PTRs). Each method will be briefly described, including it arguments, and an example (in C++) will be given. [3] Also, see Appendix B for another example in C++.

4.1 System Interface

For the following C++ examples, we will use the following C++ variables.

```
int rc;      //Return code
int trc;     //Transaction return code
```

[3] In our current implementation, the "Persist" method is actually a function but could be easily converted to a method.

Startup simply initializes an interface to the OODB, including its internal state, connection to the physical storage server, etc. Arguments could be added to allow the application program to configure the interface, include memory allocations, prefetching hints, lock type defaults, physical storage to access, etc. The return value indicates the success or failure of completing the initialization. In our implementation, this method is invoked automatically whenever an instance of Zeitgeist is allocated (as a static, automatic (stack), or pointer variable).

```
Zeitgeist azg;          //Static/Stack

Zeitgeist *zg;          //Pointer
zg = new Zeitgeist;
```

Shutdown terminates an interface to the OODB. No arguments are required. Since this method may be performed while a transaction is still pending, it must "clean up" the remains of the transaction, properly disconnect from the physical storage server, and free up any memory it may have acquired since system startup was performed. The return value indicates the success or failure of completing the termination.

```
rc = zg->shutdown(); //Manual
delete zg;           //Dynamic
```

System startup can also be performed manually after a system shutdown has been performed.

```
rc = zg->shutdown();
rc = zg->startup();
```

Begin Transaction updates the internal state of the system to record that a transaction is currently in progress. No arguments are required unless multiple-level (nested) transactions are provided. The return value could indicate the success or failure of beginning the transaction or it could be the id for the transaction just begun, if transactions other than single-level transactions are provided (to allow for proper ending of multiple-level transactions).

```
trc = zg->begin_transaction();
```

Commit Transaction indicates the end of the current transaction. No arguments are required. The system determines which objects must be saved to the database (based on modification indications), saves them, and performs whatever is necessary to end the transaction, including recording that a transaction is currently not in progress (or that the nested transaction has been completed). The return value could indicate the success or failure of committing the transaction, the time of the commit, or the number of objects committed.

```
trc = zg->commit_transaction();
```

Abort Transaction also indicates the end of the current transaction. No arguments are required. In contrast with Commit Transaction, the system performs whatever is necessary to end the transaction *without* saving any objects to the database, including recording that a transaction is currently not in progress (or that the nested transaction has been completed). The return value indicates the success or failure of aborting the transaction.

```
trc = zg->abort_transaction();
```

4.2 Individual Object Interface

Allocation of references to persistent objects (called PTRs) only requires the declaration of the references. **Deallocation** occurs when the reference leaves the variable's current language scope.

```
PTR p_ref;     // Allocate a reference
```

Assignment and Comparison operators for references to persistent objects should be specified using the same, if not similar, syntax and performed in the same, if not similar, manner as assignment and comparison of references to transient objects. At the API level, assignment and comparison of references is limited to their values, that is, the virtual memory address of the objects. Definition and implementation of deeper assignment and comparison is left to the application developer.

```
   // Allocate an instance of circuit
   // and assign reference to it.
PTR p_ref = new circuit;

   // Allocate another circuit and
   // assign another reference to it.
PTR another_ref = new circuit;

   // Compare references (i.e., addresses)
if (p_ref == another_ref)
   // Result is False

   // Assign one to the other
another_ref = p_ref;

if (p_ref != another_ref)
   // Result is False
```

Persist accepts one argument, a reference to a persistent object, and records that the application program intends that the object should become persistent (i.e., can be saved to the database as a result of the next transaction commit). The return value could either be a reference to the object argument or could indicate success or failure of this method.

```
   // Make existing object persistent
p_ref = persist(p_ref);

   // Make new object persistent
another_ref = persist( *new circuit );
```

Modification does not require any arguments and records that the application program has modified the object and intends that the object should be saved to the database as a result of the next transaction commit. If the application does not currently have a write lock on the object, the system attempts to obtain one (assuming pessimistic locking). If the lock is granted, the system records that the application program has modified this object. The return value indicates the success or failure of this method.

```
rc = p_ref.set_modified();
```

Fetch Object accepts three arguments, a reference to a persistent object, an optional lock type (read_only, read, or write), and an optional version specification. The system determines if the object has previously been retrieved (there is a cache of objects). If the object has not been retrieved, it is retrieved from the database. If a lock type is supplied, the system attempts to obtain the lock. If a version specification is supplied, the object version saved at or before the specified version (currently based on epoch time) is retrieved. The return value is the address of the persistent object in virtual memory and can only be assigned to a reference to a persistent object. A null value indicates a failure, either because the object version does not exist in the database or the requested lock could not be obtained.

```
p_ref = zg->fetch(p_ref,WRITE);

p_ref = zg->fetch(p_ref,READ,version);
```

Once a persistent object has been retrieved, persistent objects referenced from it may be retrieved implicitly (using the object fault mechanism) by simply dereferencing the reference. In the following example, assume that the class of the object referenced by p_ref has a function member 'name' which returns a reference to another persistent object whose class has a function member 'first_name'.

```
p_ref->name()->first_name();
```

When `name()->first_name()` is processed, the dereferencing of the persistent object reference returned by 'name()' will cause that persistent object to be retrieved automatically into virtual memory, if it is not already memory resident when the dereferencing occurs.

Queries on objects are orthogonal to explicit and implicit fetches and are discussed in [21].

5 Comparison with other OODBs

Table 1 presents a comparison among a small sample of OODBs in terms of the common and additional features of the API reference model they support. The systems in this comparison (some of which are in subdomains other than subdomain

2(b).ii described in the Introduction) are (in alphabetical order): Servio Corporation's GemStone [26], Hewlett-Packard's Iris [3], Object Design's ObjectStore [14], VERSANT Object Technology's VERSANT Manager/Server (formerly named OBJECT-Sciences' OBJECT-Base) [13], AT&T's ODE [10], Ontologic's Ontos [12], MCC's ORION [5], UC Berkeley's Postgres [9], and Texas Instruments' Zeitgeist [16]. Although our comparison has been based on published material publicly available to us as of this writing, we know the table may still not be accurate. We invite corrections.

In the table, N stands for No, Y stands for Yes, ? stands for unknown or unclear, and a number refers to an additional note concerning the system's API (see Section 5.1). The rows in the table describe OODBs and columns describe API features, which are numbered in the table as follows.

System Interface
1. Startup
2. Shutdown
3. Begin Transaction
4. Commit Transaction
5 Abort Transaction

Object Interface
6. Allocation/Deallocation
7. Assignment/Comparison
8. Persistent
9. Modification
10. Fetch Object

Additional Features
11. Extended Transactions
12. Grouping Objects
13. Determining and Forcing Residency
14. Fetching Previous Versions
15. Locking Objects
16. Naming Objects

As one would expect, there is a high degree of agreement among the system interface features 1 - 5. Although there is a moderate degree of agreement among the object interface features 6 - 10, the semantics of the various features are different from system to system (see Notes). And, as would also be expected, there is a low degree of agreement among the additional features 11 - 16, where the semantics widely differ. Thus, standards seem highly probable for the first set and moderately probable for the second set.

5.1 Notes on the Comparison

1. In Iris, all objects are persistent.
2. Iris and Postgres provide functions to retrieve instance values or references. Postgres also allows navigational retrieval.
3. In ObjectStore, an instance can be allocated as transient or persistent. It cannot be allocated as transient and then indicated as persistent.
4. ObjectStore allows the application developer to specify clustering only when an instance is allocated.
5. VERSANT-Manager provides "long transactions" whereby objects are checked out from a public database and copied to a private database.
6. VERSANT-Manager only allows the application developer to specify clustering **prior** to adding objects to the database. Physical clustering of aggregates is not guaranteed. The default clustering is of instances of the same class.
7. ODE defines two new operators (pnew and pdelete) to allocate/deallocate a persistent object. A transient object can be copied to a newly created persistent object but it cannot be designated as persistent.
8. ODE allows the user to fetch objects using a query specifying an instance's value. It is not clear whether they allows fetches using a reference to a persistent object.
9. ODE allows the application to dynamically define object clusters. However, the clusters can only contain instances of one specific type.
10. ODE allows object versions, but a new construct is defined to reference versions of an object and create/delete a version.
11. Ontos has two kinds for references to persistent objects. The mechanism used to fetch and store persistent objects depends on the kind of reference used.
12. In Orion, instances of classes derived from Orion's base class cannot be transient.
13. Orion only allows the application developer to specify clustering **prior** to adding objects to the database. The cluster can contain instances

System	System Interface					Object Interface					Additional Features					
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GemStone	Y	Y	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	N	Y	Y
Iris	Y	Y	Y	Y	Y	?	?	1	1	2	?	Y	?	?	?	?
ObjectStore	Y	Y	Y	Y	Y	N	Y	3	N	?	N	4	?	Y	?	N
VERSANT-Mgr	Y	Y	Y	Y	Y	Y	Y	Y	?	Y	5	6	?	Y	Y	?
ODE	Y	Y	Y	Y	Y	7	Y	7	?	8	?	9	?	10	?	N
Ontos	Y	Y	Y	Y	Y	Y	Y	Y	N	11	Y	Y	Y	Y	Y	Y
ORION	Y	Y	Y	Y	Y	Y	Y	12	?	Y	Y	13	?	14	Y	?
Postgres	Y	Y	Y	Y	Y	?	?	?	Y	1	?	?	?	Y	Y	N
Zeitgeist	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	15	Y	Y	Y	Y

Table 1: Comparison among a representative sample of OODBs.

of one specific class or instances of a user-specified collection of classes.

14. In Orion, only instances of classes which have been declared to be *versionable* can be versioned.

15. Zeitgeist allows the application developer to specify static clustering when classes are defined and dynamic clustering when indicating persistence.

6 Conclusions

This paper has proposed a reference model for comparing application program interfaces to Object-Oriented Databases, identified areas where consensus is possible and standards can emerge, and areas where further research is needed before a consensus can be reached.

We have proposed an API which provides a better integration of database and programming languages by adhering as closely as possible to the structure and intent of the host programming language (C++ or CLOS) without unduly changing or extending the language. In addition, the API promotes a philosophy of "only paying for what you want when you want it" to provide the various features presented, especially with respect to allocation of objects, indication of persistence, indication of modification, and actual saving of objects.

We believe that work on standards for APIs to OODBs needs to be actively pursued by standards committees in parallel with the current efforts in other aspects of object-orientation. Such committees include the X3/SPARC/DBSSG/OODBTG on a reference model for object databases, Object Management Group (OMG) on an object software framework, X3J16 on C++, and X3J13.1 on CLOS.

Appendix

A Data Definition Language

In order to define types and classes to the Zeitgeist OODB, we have developed a Data Definition Language (DDL), which is the C++ language plus a few extensions and a few restrictions, and a translator to process the language. The translator generates C++ .h files as well as information for the Zeitgeist runtime system.

The extensions allow the class developer to 1) identify the kind of persistence an object will have, 2) provide information on the boundary of a persistent object graph, and 3) define demons (functions invoked at specific time during an application program execution). We will discuss the first two extensions in the next two subsections.

A.1 Persistent Objects

In order for a object of a class to be saved to the database, its class definition must be processed by our DDL translator; objects of these classes are called potentially persistent, or persistent, for short. Any class not processed by our translator cannot have its objects saved to the database; objects of these classes are called fully transient, or transient, for short.

Persistent objects can be stored either **independently** or **dependently**. The designation is made in the DDL and is one of our extensions to C++. Independent persistent objects become the root of a graph of objects. They can reference any other

object (persistent or transient) and can be referenced by one or more other objects (persistent or transient). Dependent persistent objects can also reference any other object (persistent or transient) but can only be referenced from ONE persistent object and/or one or more transient objects. In other words, a object that is shared (referenced from two or more other persistent object) **must be designated as independent.**

Figures 1 and 2 show examples of dependent and independent class definitions.

A.2 References to Objects and Data

As stated above, transient and persistent objects can reference other transient and persistent objects. References to transient objects are accomplished by using C++ pointers. References to persistent objects can be accomplished by using C++ pointers or Zeitgeist PTRs. In either case, if a C++ pointer is used and the referenced object is not to be saved, the keyword "boundary" (another of our extensions) must precede the declaration. If a C++ pointer is used and the reference object(s) is(are) to be saved, a sentinel (another of our extensions) must be specified to allow Zeitgeist to determine the number of objects actually referenced (since C++ pointers may be used to point to an array of objects). No language extensions are necessary when a Zeitgeist PTR is used since, at this time, only one object can be referenced via the PTR.

Figure 3 shows an example of a class which has references to objects of other classes (it could reference another object of its own class). Assume that the class Class_Object is a transient class, the class Instance_Object is a dependent persistent class, and the class Schema is an independent persistent class.

The Class_Object referenced via 'has_class' will NOT be saved due to the keyword "boundary". The Instance_Object and Schema referenced via 'dummy_instance' and 'dummy_schema', respectively, will also NOT be saved due to the keyword "boundary" (even though objects of those two classes can be saved to the database). Only one Instance_Object referenced via 'first_instance' will be saved (there are other extensions to dynamically determine the number of referenced objects to be saved). The OID of the Schema referenced via 'in_schema' will be saved. That Schema will be saved separately from this object.

C++ fundamental types and structs can also be referenced using C++ pointers. As with references to objects, if the data is to be saved, a sentinel must be supplied to allow Zeitgeist to determine the number of elements referenced. If the data is not to be saved, the keyword "boundary" must be used as described earlier.

A.3 Other Issues

We have only covered two issues in this section due to reasons of space and scope. Other issues which should be explored in the context of data definition include demons or triggers (functions which are invoked at specific times during application program execution), implementation of relationships (including sets, hash tables, collections, etc.), and processing of class libraries to allow them to be persistent.

B Examples

Figure 4 (4 pages) shows an example program written in C++ interfacing to Zeitgeist.

C Glossary

Below is a short list of terms from the fields of programming, object-oriented programming, and databases. We refer readers to the glossary of terms being compiled by the OODB Task Group for a more complete list along with definitions.

application programming language

database language

database programming language

persistent language

data model

type system

object database

object base

inheritance

generalization

specialization

abstract data type

encapsulation

data abstraction

metaclass
class (type)
subclass (derived class)
superclass
object
object identity
object identifier
unique object identifier
object name
object reference
object retrieval
object fault
instance
transience
persistence
transient class
persistent class
transient object
persistent object
shallow equality
deep equality
composition
aggregation
complex object
relationship
link
attribute
property
data member
instance variable
slot
behavior

function
member function
method
message
message passing
read-only operation
read/write operation
concurrency
integrity
transaction
commit transaction
abort transaction
cluster
object cluster
version
object version
time stamp
lock
object lock
lock conflict
lock conflict resolution

References

[1] Lyngbaek, P., Kent, W. A Data Modeling Methodology for the Design and Implementation of Information Systems. In *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sep. 1986.

[2] Fishman, D. H., Beech, D., Cate, H. P., Chow, E. C., Connors, T., Davis, J. W., Derrett, N., Hoch, C. G., Kent, W., Lyngbaek, P., Mahbod, B., Neimat, M. A., Ryan, T. A., and Shan, M. IRIS: An object-oriented database management system. *ACM Transactions on Office Information Systems 5*, Jan. 1987.

[3] Wilkinson, K., Lyngbaek, P., Hasan, W. The Iris Architecture and Implementation. *IEEE Trans. on Knowledge and Data Engineering*, Mar. 1990.

[4] Kim, W., Ballou, N., Banerjee, J., Chou, H.-T., Garza, J., Woelk, D. Integrating an Object-Oriented Programming System with a Database System. In *Proc. 1988 3rd Int. Conf. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, San Diego, CA, Sep. 1988.

[5] Kim, W., Garza, J., Ballou, N., Woelk, D. Architecture of the ORION Next-Generation Database System. *IEEE Trans. on Knowledge and Data Engineering*, Mar. 1990.

[6] Richardson, J. E., Carey, M. J. Programming Constructs for Database System Implementation in EXODUS. In *Proc. 1987 ACM SIG on Management of Data (SIGMOD)*, San Francisco, CA, May 1987.

[7] Stonebraker, M. Object Management in POSTGRES using Procedures. In *Proc. 1986 Int. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sep. 1986.

[8] Rowe, L. A., Stonebraker, M. The POSTGRES Data Model. Memorandum UCB/ERL/M86/85, University of California, Berkeley, Berkeley, CA., Jun. 1987.

[9] Stonebraker, M., Rowe, L. A., Hirohama, M. The Implementation of POSTGRES. *IEEE Trans. on Knowledge and Data Engineering*, Mar. 1990.

[10] Agrawal, R., Gehani, N. H. ODE (Object Database and Environment): The language and the data model. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, OR, May 1989.

[11] Maier, D., Stein, J. Development and Implementation of an Object-Oriented DBMS. Computer Science Series. The MIT Press, Cambridge, MA, 1987.

[12] Andrews T., Harris C., Duhl J. Ontos Object Database. Ontologic Inc., Burlington, MA, 01803, Mar. 1990.

[13] Object-Sciences Corp. Object-Sciences System Description Manual. Object-Sciences Corp., Menlo Park, CA,, Mar. 1990.

[14] Object Design. An Introduction to ObjectStore, Release 1.0. Object Design, Burlington, MA,, Mar. 1990.

[15] Ford, S., Joseph, J., Langworthy, D. E., Lively, D. F., Pathak, G., Perez, E. R., Peterson, R. W., Sparacin, D. M., Thatte, S. M., Wells, D. L., and Agarwala, S. ZEITGEIST: Database Support for Object-Oriented Programming. In *Advances in Object-Oriented Database Sytems, 2nd International Workshop on Object-Oriented Database Systems*, Bad Münster am Stein-Ebernburg, FRG, Sep. 1988.

[16] Perez, E. ZEITGEIST Persistent C++ User Manual. Technical Report 90-07-02, Information Technologies Laboratory, Computer Science Center, Texas Instruments, Inc., Dallas, TX, Jun. 1990.

[17] Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S. The Object-Oriented Database System Manifesto. ALTAIR Technical Report No. 30-89, GIP ALTAIR, LeChesnay, France, Sep. 89.

[18] Stonebraker, M., Rowe, L. A., Lindsay, B., Gray, J., Carey, M., Beech, D. Third Generation Data Base System Manifesto. The Committee for Advanced DBMS Function, Mar. 1990.

[19] Kim, W. Research Directions in Object-oriented Databases. MCC Technical Report ACT-OODS-013-90, MCC, Austin, TX, Jan. 1990.

[20] OODB Task Group. Reference Model for Object Data Management, OODB Task Group Document OODB 89-01R4, ANSI/X3, DBSSG/OODBTG, August 1990.

[21] Blakeley, J., Thompson, C. W., Alashqur, A. M. Strawman Reference Model for Object Query Languages. Technical Report, Information Technologies Laboratory, Computer Science Center, Texas Instruments, Inc., Dallas, TX, Jun. 1990.

[22] Joseph, J., Shadowens, M., Chen, J., Thompson, C. W. Strawman Reference Model for Change Management of Objects. Technical Report 90-06-02, Information Technologies Laboratory, Computer Science Center, Texas Instruments, Inc., Dallas, TX, Jun. 1990.

[23] Wang, C. C. Strawman Reference Model for Transactions. Technical Report, Information Technologies Laboratory, Computer Science Center, Texas Instruments, Inc., Dallas, TX, Aug. 1990.

[24] Khoshafian S., Copeland G. Object Identity. In *Proc. 1986 1st Int. Conf. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Portland, OR, Sep. 1986.

[25] Abiteboul S., Kanellakis P. Object Identity as a Query Language Primitive. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, Portland, OR, May 1989.

[26] Maier D., Stein J., Otis A., Purdy A. Development of an Object-Oriented DBMS. In *Proc. 1986 1st Int. Conf. Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Portland, OR, Oct. 1986.

```
class Pin
   // no "persistent" indicates Dependence
{
private:
  long    instanceLevel;
  long    currency;
  String  pinNumber;
  String  secondaryPinNumber;
  PinType pinType;
  double  inverseInductance;
  double  resistance;

public:
        // NOTE: NO PUBLIC DATA MEMBERS
  Pin();
  ~Pin();
  // other functions would go here
};
```

Figure 1: DDL for Dependent Persistent Class

```
class Cell : persistent
   // "persistent" indicates Independence
{
private:
  String   cellName;
  String   cellAliasName;
  CellType cellType;
  long     numberTransistors;
  String   propagationDelay;
  long     dcCurrent;
  double   cPd;
  double   xDim, yDim;

public:
        // NOTE: NO PUBLIC DATA MEMBERS
  Cell();
  ~Cell();
  // other functions would go here
};
```

Figure 2: DDL for Independent Persistent Class

```
class Schema_Collection : persistent
{
private:
  boundary
  Class_Object    *    has_class;

  boundary
  Instance_Object *    dummy_instance;

  boundary
  Schema          *    dummy_schema;

  Instance_Object *[1] first_instance;

  SchemaPTR            in_schema;

public:
        // NOTE: NO PUBLIC DATA MEMBERS
  Schema_Object();
  ~Schema_Object();
  // other functions would go here
};
```

Figure 3: DDL for References

This is an example C++ program which interfaces with the Zeitgeist OODB.

Figure 4
pg. 1

```
#include <stdio.h>

#include <zeitgeist.h>

#include "example.h"
#include "example_i.h"


#define  DEFAULT_LIST  "OODBTG"

//  Globals

zeitgeist zg;                                        //<<<< ZG code


void main (int argc, char* argv[])
{
  int i, nobjs=0, count=0, firsttime=0;

  examplePTR headptr, *nextp, prevptr;    // for adding
  examplePTR fetchptr;                    // for traversing


  //  Create new name context

  zg.begin_transaction ();                           //<<<< ZG code
  zg.default_name_context ("test prog");             //<<<< ZG code
  zg.commit_transaction ();                          //<<<< ZG code


  //  Get number of links for new list

  nobjs = atoi (argv[1]);
  printf ("Append %d links to link list \"%s\"\n\n", nobjs, DEFAULT_LIST);


  // Build the list

  zg.begin_transaction ();                           //<<<< ZG code

  nextp = &headptr;
  for (i=0; i<nobjs; i++)
    {
      *nextp = persist (*new example);
      prevptr.set_modified ();                       //<<<< ZG code

      (*nextp)->prev () = prevptr;
      prevptr = *nextp;
      nextp = &((*nextp)->next ());
    }


  //  Save the list

  if (headptr.name (DEFAULT_LIST) < 0)               //<<<< ZG code
    {
      printf ("%s: error naming list root object\n", argv[0]);

      zg.shutdown ();                                //<<<< ZG code
```

```
    exit (-1);
  }

if ((nobjs = zg.commit_transaction ()) < 0)    //<<<< ZG code
  {
    printf ("%s: error committing transaction\n", argv[0]);

    zg.shutdown ();                            //<<<< ZG code
    exit (-1);
  }


//  Fetch head of list then display all links in the list

printf ("Fetch head of list \"%s\" for display\n\n", DEFAULT_LIST);

zg.begin_transaction ();                       //<<<< ZG code

if ((fetchptr = zg.fetch (DEFAULT_LIST)) == NULL)     //<<<< ZG code
  {
    printf ("could not fetch object \"%s\"\n", DEFAULT_LIST);

    zg.shutdown ();                            //<<<< ZG code
    exit (-1);
  }

nobjs = 0;
while (fetchptr != NULL)
  {
    nobjs++;
    prevptr  = fetchptr;
    fetchptr = fetchptr->next ();              //<<<< Object faulted
  }


//  Display the list backwards via the back pointers in the list

printf ("\nNow, in reverse.\n\n");

while (prevptr != NULL)
  {
    prevptr = prevptr->prev ();
  }

zg.abort_transaction ();                       //<<<< ZG code


//  Shut down Zeitgeist and exit test

zg.shutdown ();                                //<<<< ZG code
printf ("\nTest complete.\n");
exit (0);
```

Figure 4
Pg. 2

This is the DDL description for class "example".

Figure4
pg.3

```
class example : public x
{
public:

  //  constructor & destructor

  example ();
  ~example ();

  //  methods

  examplePTR& prev ();
  examplePTR& next ();

private:

  //  state

  char line[48];
  examplePTR prev_link;          // Use ZG PTR's instead of C++ pointers (*)
  examplePTR next_link;          // to reference persistent instances

};
```

Here is the code for the methods.

```
example::example ()
{
  prev_link = NULL;
  next_link = NULL;
}

example::~example ()
{ // null body }


examplePTR& example::prev ()
{
  return (prev_link);
}


examplePTR& example::next ()
{
  return (next_link);
}
```

This is the DDL description for class "x".

```
class x : public persistent
{
public:

  //  constructors & destructors

  x ();
  ~x ();

protected:

  //  state

  char *[48] xp1;                  // Specify length of character string
  char *[48] xp2;                  // referenced from an instance of x
  char *[48] xp3;                  // using "[48]" extension.
};
```

Figure 4
pg. 4

Here is the code for the methods.

```
x::x ()
{
  int size = 48;

  //  Initialize the x object

  xp1 = new char[size];
  xp2 = new char[size];
  xp3 = new char[size];

  sprintf (xp1, "class x member xp1");
  sprintf (xp2, "class x member xp2");
  sprintf (xp3, "class x member xp3");



::~x ()

  delete xp1;
  delete xp2;
  delete xp3;
```

An Approach to Standard DDL for OODBMSs

by
Peter Moore, peter@objy.com
Andrew E. Wade, drew@objy.com
Objectivity, Inc.

Presented to:
ANSI SPARC DBSSG OODBTG Workshop
Chateau Laurier Hotel, Ottawa, Canada
Oct. 23, 1990

ABSTRACT

In this paper, we focus on standards for an OODBMS Data Definition Language (DDL), a facility to specify object classes. Key requirements include support of multiple languages, multiple and large databases, portability and heterogeneity, avoiding definition of yet another language, and support of both dynamic and pre-compiled interfaces. We propose an approach based on C++, and address semantics necessary to provide the following capabilities: well-defined primitives for portability and heterogeneity, varying-sized arrays, associations (links), versions, and propagation of methods among objects. We conclude it is possible to converge on this DDL standard, allowing object models to be defined once for multiple DBMSs.

CONTENTS

1. Introduction and Motivation

In Objectivity's presentation to the last OODBTG workshop[1], we discussed the motivation for standards in OODBMSs, stressing the importance of recognizing and agreeing on such a motivation. We proposed the motivation is *interoperability*: the ability to mix and match a variety of applications and DBMSs.

To achieve interoperability, we listed several areas that could be standardized, focusing on the interfaces, and leaving as much freedom as possible to the implementors to further the state of the art. These areas include: Object Model, Data Definition Language (DDL), Data Manipulation Language (DML), Query, Class Libraries, Interchange, and others. By addressing the first three, and specifically capabilities in the DDL and DML areas, we can begin to address the needs of applications and provide some measure of interoperability. In this paper we present some of the issues and some suggested directions for standardizing DDL.

2. Environmental Requirements of DDL

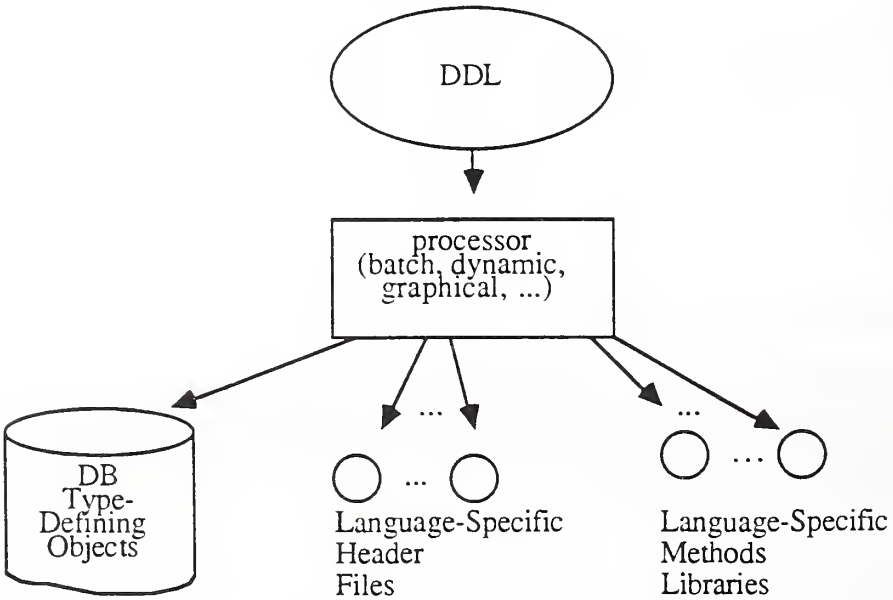
We can make several assumptions about the environment in which the DDL architecture must live. Eventually, the actual DDL standard must include specific syntax. However, in this paper, we focus on semantics. Our use of syntax should be considered for the purposes of illustration only.

The DDL standard should address both dynamic (runtime type definition/modification) as well as a pre-compiled language interface. The standard may even eventually include graphical notation. All of these should simply be different means to the same end, whose semantics we address here. For simplicity and concreteness, we illustrate concepts with a language presentation.

Architecturally, we assume:

- the DBMS will support the concept of a class or type (used interchangeably here)
- instances of these types may be instantiated
- the DBMS understands these type definitions, presumably as type-defining objects, or the equivalent
- the standard will support multiple databases, large databases, and multiple languages

This last assumption leads to a conceptual model as shown in Figure 1.



Multiple Language Support
Figure 1

Note that the conceptual model can be applied to any implementation: dynamic, batch, graphical, etc. The model illustrates the concept of a standard DDL specification that is processed somehow to produce:

- the type-defining objects with the DBMS
- header files for various languages
- libraries of methods for various languages

The DDL standard should support these environmental and architectural requirements, and should support a variety of vendor implementations of the conceptual structure shown in Figure 1.

3. Leverage C++

One of our requirements is to avoid creating a new language for DDL. Instead, we propose using C++ as a starting point. [Another choice worth considering is STEP/Express[2].]

C++ provides the capabilities to specify a large variety of data structures, encapsulated methods, and inheritance. The class definition facility of C++ allows classes to be added to extend the C++ standard. This facility allows us to specify:

- any C or C++ data structure
- any arbitrary method than can be expressed in C or C++
- the inheritance hierarchy

The resulting DDL will be familiar to C++ users and largely familiar and understandable to C users. So, not only do we save the effort of defining a new language, but we can hope to achieve wider and faster adoption by users.

4. Additional Requirements

In this section, we discuss DDL requirements not provided by C++.

4.1 Well-Defined Primitives

The structural part of the object definition in DDL should be well-defined to allow support for portability and heterogeneity. Terms such as *short* and *long* can lead to confusion when different system environments interpret them differently: are they 16 bits or 32 or 64, signed or unsigned?

To avoid this problem we suggest the standard DDL define specific primitives. Here are some possible examples.

Primitive	Range
uint8	0..255
int16	-32768..32767
uint32	0..4,292,967,295
etc.	

Well-Defined Primitives
Figure 2

In addition the standard can specify what the default interpretation is for generic types, as shown in Figure 3.

Generic C++ Type	DDL specific type
short	int16
long	int32
float	float32
etc.	

Correspondence of Generic C++ Types
Figure 3

4.2 Varying-Sized Arrays

Data structures that dynamically vary in size are commonly used by applications. However, these structures are not provided in C++. Because of their importance, we suggest that these structures be incorporated into the DDL standard as a class extension to C++. We suggest adding the concept of a *VArray* ; a varying-sized array of arbitrarily elements, providing a generic and powerful mechanism to specify arbitrarily varying-sized structures. The DBMS should provide memory management facilities and DML access facilities to support VArrays.

Figure 4 shows the DDL specification for VArrays. When C++ parameterized classes become available, we can replace the VArray(Class) macro with the proposed syntax VArray <Class>.

```
struct Point              {
   int32                  x, y;
};

struct Path :      Geometry   {
   int32                       width;
   VArray (Point)  points;
}
```

DDL Specification of Varying-Sized Array
Figure 4

4.3 Associations or Links

In programming languages, the ability to use a pointer construct as a structure primitive provides users with a broad range of capabilities. The same capabilities are required in the OODBMS; i.e., there is a need for direct, inter-object links, with the ease and efficiency of pointers. However, pointers themselves are limited to the context of a particular process, a particular virtual memory address space, and particular hardware architectures. Using plain OIDs instead of pointers ties the user model too closely to the physical model, entailing the problems encountered in Codasyl Network model.

In order to support portability across multiple architectures, to support heterogeneity or simultaneous use of multiple architectures, and to support large databases without the limit of virtual memory address spaces, we propose a specific *association* construct in the DDL. Similar concepts exist in several systems: e.g., the PCTE link[3,4]; many engineering database systems, such as Mentor's inter-object *References* [5]; and in the Storage Manager standard being produced by the CFI[6].

Associations go beyond the traditional language construct of pointers to include some often-used database functionality. In particular, the association can be uni- or bi-directional. In the latter case, the semantics include maintaining referential integrity. Associations can be one-to-one, one-to-many, or many-to-many to support common application needs. Like pointers, associations should also support strong type checking to a level specified by the user.

Access to associations is the domain of DML, but we suggest it resemble the use of pointers. The concept of *iterators* can be introduced into the DML to allow traversal of many-to-many associations.

To illustrate the concept of associations in the DDL notationally, we represent the association as a member field in a class, with a type represented by the macro Handle(Class). Handle is simply a way to indicate an association, while Class provides the binding necessary for type checking. As with VArray, when C++ parameterized classes become available, we can replace the Handle(Class) macro with Handle <Class>.

```
class Layer :   ParentClass                {
...
Handle(Cell)          cell <->  layers[];
Handle(Shape)   shapes[]  <->  layer;
}

class Shape :   ParentClass                {
...
Handle(Layer)   layer     <->  shapes[];
}
```

DDL Association Example
Figure 5

Notice that associations appear much like normal pointers, for compatibility with typical programming style and readability. The bi-directional arrows <-> indicate bi-directional associations. (A single arrow -> would indicate a uni-directional association.) Finally, the brackets [] indicate a many-side of the association; e.g., the layer-to-shapes relationship has many shapes per layer.

4.4 Versioning

The need for versioning of objects is often discussed within DML, outside the domain of DDL. However, specifying versioning-related behavior is appropriate at the DDL level. To do so, we first introduce a general extension to associations as shown in Figure 6.

```
Handle(Class)   linkName   :bspec, bspec;
```

Association Behavior Specification
Figure 6

Here, *bspec* is a generic *behavior specification* . One or more behavior specifications can be used.

Now, consider the effect of versioning associated objects. What does the DBMS do to the association when the linked objects are versioned? With a *behavior specification*, we can allow the standard DDL the flexibility to control this. For example, the versioning behavior of associations can be specified as shown in Figure 7.

```
Handle (Class)  linkName   :version (copy);
```

Versioning Behavior Specification
Figure 7

Here, *copy* indicates that the association link should be copied to the new version, so both old and new remain linked. Other possible versioning behaviors include *move*, move the link to the new version, and *drop*, drop the association link altogether. Similarly, we could specify versioning behavior as none, linear, or branching.

4.5 Propagation of Methods

What about methods? They can be specified within classes, but how does their behavior interact with associations? We can allow the DDL to specify this by introducing the concept of *propagation of methods* along associations. A simple example would be to propagate the delete method across associated objects. Then, the user could conceptually treat the entire collection of associated objects as one *composite object* and apply methods to it as a whole. The behavior specification can serve this need (see Figure 8).

```
Handle (Class)     linkName   :propagate (op,op,op);
```

Propagation of Methods
Figure 8

Here, the *propagate* behavior specification indicates that specified operations, when applied to object instances of this class, will also apply to associated objects.

5. Future Areas

Many other areas of capability could be discussed; e.g., specification of indexes, modification of type definitions, hints for storage management (e.g., clustering, typical or expected sizes), constraints (e.g., minimum and maximum cardinality of associations), security, etc. We suggest that the behavior specification (introduced above for versioning and propagation) could be applied to accommodate many such capabilities. Similarly, the fundamental ability to extend C++ by adding classes (used above for well-defined primitives and VArrays) could be applied to accommodate many capabilities. So, while we make no claim that we have exhaustively covered the areas of need in DDL, we suggest this approach can be applied and extended.

6. Conclusion

To achieve standardization of OODBMSs, and in particular to allow applications to be written against this standard and to be run against multiple vendors *(interoperability)*, both the DDL and DML need to be established standards. We have presented a proposal for DDL standardization.

Our proposal is based on these environmental requirements:

- multiple languages
- agreed semantics
- support for large databases, multiple databases
- support for portability and heterogeneity
- support for dynamic, pre-compiled, and graphical interfaces

The content of our proposal can be summarized as follows:

- Based on C++ class definitions for data structure, methods, and inheritance, with specifications added for:
- Well-defined primitives.
- Dynamically varying-sized arrays
- Associations of links, for inter-object relationships
- Versioning
- Propagation of methods

We invite comments, reactions, criticism, and additions to these suggestions.

References

[1]Guzenda, L., "Taxonomy of Standards: Position Paper", OODBTG Workshop Proceedings, May 22, 1990.
[2]Schenck, D., "Information Modeling Language Express", ISO Doc. N287, Oct. 31, 1988, available from NIST as part 2 of PDES, P889-144794.
[3]"Introducing PCTE+", Independent European Programme Group, TA-13 document, GIE Emeraude.
[4]"Rationale for the Changes Between the PCTE+ Specification", Issue 3, Jan. 6, 1989.
[5]Ecklund, E., Mentor Falcon CFI presentation, Feb. 28, 1990.
[6]Wade, A., ed., Storage Manager Goals and Requirements, TCC-Approved Draft Proposal, Cad Framework Initiative, V0.6, Aug. 2, 1990.

SECURITY STANDARDS FOR OBJECT DATA MANAGEMENT SYSTEMS

Victoria Ashby
Linda Schlipper
The MITRE Corporation
McLean, Virginia

INTRODUCTION

Object data management (ODM) systems represent a rapidly emerging technology which combines object-oriented programming concepts with database management technologies. The object paradigm is attractive for databases because it supports complex data structures, inheritance, and the behavioral characterization of data. The result is a semantically rich data environment for representing structure and data in real-world applications. ODM technology has now reached the point where formal standardization has become an option. The Object-Oriented Database Task Group (OODBTG), a task group of the ANSI/SPARC Database System Study Group (DBSSG), has issued a draft *Reference Model for Object Data Management* (OODBTG, 1990) and begun to hold workshops to determine what aspects of these systems should be included in the standardization process. This paper is in response to a request by the OODBTG for recommendations for ODM system standardization activities.

The authors of this paper take the position that security features should be part of the emerging standards for the ODM model. Security features are important components of ODM systems since the protection of data from unauthorized or unintended disclosure, modification, or destruction is an important concern in both the commercial and government sectors. Moreover, early consideration of security features is important because experience in the relational database management systesms (RDBMS) and operating system areas has shown the difficulty of attempting to retrofit security to an established technology. Although RDBMSs have dominated most of the work on database security (Hinke, 1988; Lunt, 1988; Stachour, 1990), security issues in ODM systems have also begun to be investigated (Fernandez et al, 1989; Keefe et al, 1989; Jajodia and Kogan, 1990; Lunt and Millen, 1989; Thuraisingham, 1989a, 1989b, Thuraisingham and Chase, 1989). Indeed, initial work with the ODM model indicates that it has many features, such as encapsulation, object identity, and class hierarchies, that naturally support security concepts.

This paper examines the incorporation of security features within an ODM system, and addresses the importance of incorporating certain of these features in the current standardization process. In the next section we provide an overview of concepts important to information security, including access control, security models, secrecy constraints, roles, and assurance. Following this, we identify ODM model features that have important implications for the support of these security concepts. We conclude by discussing security features appropriate for inclusion in the ODM system standardization process.

OVERVIEW OF INFORMATION SECURITY CONCEPTS

This section gives a brief overview of information security concepts fundamental to this paper. We begin with a definition of information security. This is followed by a discussion of the use of access controls as a way to protect data from unauthorized disclosure and also from unauthorized modification. Next, the concept of a security model, which identifies permitted access modes in accordance with the specific system security policy, is described. Secrecy constraints are introduced as a way to extend security policies for database systems. The notion of a user role as a control mechanism for access privileges is then explained. The section concludes with a discussion of assurance, a way of confirming that technical protections within a computer system provide security.

Information Security

Information security consists of three components: secrecy, integrity, and availability. *Secrecy* addresses the need to protect information from unauthorized disclosure. Even in the least secure system, users may wish to ensure the confidentiality of their data against disclosure to other users. *Integrity*, as traditionally discussed in the information security literature, addresses the need to protect information from unauthorized modification. In a broader context, integrity refers to protecting information against inappropriate modification; that is, modification that does not correspond to the real world, whether the modification is authorized or not. *Availability* addresses protection against denial of service. Denial of service can occur when a malicious user, legitimate or not, ties up resources, making them unavailable to other legitimate users. In the past, secrecy has been emphasized in security work, while integrity has been important in database management system (DBMS) work. Availability, an extremely complex problem, is only beginning to be addressed (Gasser, 1988).

Access Control

Because controlling access to data provides a way to protect against both unauthorized disclosure and unauthorized modification of information, access control and types of access privileges have been the focus of much work by the security community.* There are two types of access control in a secure system. The first, discretionary access control, allows access to data to be specified and controlled at the user or user group level. Discretionary access control commonly includes the idea of control of access by the data owner. The usual approach to controlling discretionary access in a database involves access matrices, which specify which operations a user (or group) is authorized to perform on some set of data. The second type of access control is called mandatory access control. Mandatory access control restricts access to information based on the sensitivity of the information (e.g., classification) and the extent to which a user is trusted to access that information (e.g., clearance). In this context, data is labeled with classification levels or sensitivity levels, while users have clearance levels that reflect the degree to which they are authorized to access data. The clearance and classification levels together comprise an access class. Mandatory access control is a necessary component of multilevel

---

* Being authorized to modify data does not guarantee that the modifications maintain integrity.

secure (MLS) DBMSs, which store data labeled with different sensitivity levels in one logical database. The database is accessed by users in sessions at different classification levels. The MLS DBMS is trusted to separate user sessions so that each user sees data appropriate to his or her session level, and nothing else.

Security Models

A computer system is considered secure when it meets the security requirements defined for that system. Those security requirements dealing with access control are expressed in a security model, which is an informal, natural language statement or a formal, mathematical model. The most widely referenced security model is the Bell and LaPadula model (Bell and LaPadula, 1973). This model states which subjects in a system can access which objects, and consists, informally, of the following two properties: a subject can read an object if the subject's clearance level dominates the sensitivity level of the object (simple security property), and a subject can write to an object if the subject's clearance level is dominated by the objects sensitivity level (*-property). A system state is "secure" if the only permitted access modes of subjects to objects are in accordance with these two properties. The system, itself, is secure if each transition between states results in the return to a secure state. If there are ways to access data that are outside of the security model, these are called *covert channels*.

Secrecy Constraints

Secrecy constraints address additional authorization and classification issues that the discretionary and mandatory access controls described above do not address. These issues include content-based and context-based access control, association constraints, and aggregation constraints. Content-based access control refers to the granting or denying of access based on attribute values (i.e., content of the data element). A familiar mechanism for controlling content-based access is views in RDBMSs. Context-based access control needs arise when two or more attributes may be accessed together. For example, it may be desirable to restrict access to employee names when they are retrieved with their salaries although values of either attribute may be viewed separately. Association constraints classify the relationships between entities in the database. An example of such a constraint would restrict the association of employees and certain sensitive projects on which they work Finally, aggregation constraints deal with the problem that some aggregates of data have a higher sensitivity level than the individual components.

User Roles

The least privilege principle requires that each subject (e.g., user or application process) in a system is granted only the most restrictive set of privileges needed for the performance of its authorized task, thus limiting the damage that can result from accident, error, or unauthorized use. This principle corresponds to the idea of allowing a database administrator or a database designer to have a different set of privileges than an end user. A *role* allows an individual user account to take on privileges associated with specific aspects of system administration or application processing. This supports not only individual accountability, but also the least privilege principle by separating the functions performed by a security administrator and a database administrator.

Assurance

Assurance means that a system has the qualities of correctness and tamperproofness. Assurance is important for any secure system, but it is particularly important for MLS systems. The MLS approach combines technical protection within a computer system with physical and procedural protections outside of a computer system. The functionality of an MLS system is fairly easy to provide; however, the functions must be built securely to provide technical protection to the system. It is the need for strong assurance that makes MLS capabilities difficult to achieve. As discussed in the next section, the ODM model provides both the possibility of adding security features and the possibility of increased assurance to support multilevel database security.

ODM SYSTEM FEATURES THAT SUPPORT SECURITY CONCEPTS

The draft *Reference Model for Object Data Management* (OODBTG, 1990) contains a section on security which states that "the security facilities should be integrated with the concepts of encapsulation, objects, inheritance, and identity." Many researchers in database security believe that the ODM model offers real advantages for the support of security. In this section, we identify those features of the model that we feel naturally support security concepts: behavior, classes hierarchies, encapsulation, and object identity.

Behavior

In an ODM system, all conceptual entities are objects, each object consisting of a set of named attributes and a set of operations that can access those attributes. This set of operations, or methods, describe the object's behavior. This ability to model behavior with methods provides new semantic facilities for imposing constraints on the relationships between data. A number of researchers have looked at methods as a way of augmenting ODM systems for secrecy constraints. Lunt and Millen (Lunt and Millen, 1989) and Thuraisingham (Thuraisingham, 1990) have identified approaches for content- and context-based constraints for both discretionary and mandatory access controls. Thuraisingham also identified aggregate constraints as a mechanism for addressing classification aggregation problems. In addition, Lunt and Millen looked at constraints as a way of handling the three "dimensions" of classification identified by Smith (Smith, 1988). Lastly, encoding inference rules in methods has been suggested by Lunt and Millen as a partial solution to inference problems.

Class Hierarchies

Classes are groupings of objects which share similar characteristics: attributes or methods. The concept of a class hierarchy extends the class concept further by allowing the attributes and methods specified for one class to be inherited by its subclasses, thereby leading to less redundancy in data and method implementation. Inheritance also simplifies data classification by allowing security constraints to be attached to classes and inherited by their subclasses. Likewise, it allows the existing structure of the system to be used as the basis for multilevel object classification. Jajodia and Kogan (Jajodia and Kogan, 1990) use the term *security inheritance* to describe the representation of multilevel entities in an ODM system using class hierarchies.

Encapsulation

Another important feature of ODM systems is the notion of encapsulation. Object encapsulation means that an object can only be acted on by executing the methods defined for that object. This feature supports the concept of a reference monitor. A reference monitor is defined by the National Computer Security Center's (NCSC) *Trusted Computer System Evaluation Criteria* (TCSEC) (NCSC, 1985) as "an access control concept that refers to an abstract machine that mediates all accesses to (security) objects by subjects." An example of how object encapsulation can be applied to the reference monitor concept is provided by Jajodia and Kogan (Jajodia and Kogan, 1990). Since all constructs are objects and all objects must communicate by messages, they say that "messages can be considered the only instrument of information flow" in an ODM system. They go on to propose that every message can be "intercepted by the message filter, a system element charged with implementing security policies." This design enforces the reference monitor concept because everything is an object, all objects communicate through messages, and all messages go through the security filter.

Object Identity

Object identity addresses polyinstantiation, which is a phenomenon that occurs in secure RDBMSs to correct a covert channel problem. Polyinstantiation refers to the simultaneous existence in a relation of multiple tuples with the same key values, where the multiple instantiations are distinguished by their security levels. The existence of polyinstantiated tuples leads to a referential integrity problem during updates. However, there are no duplicate keys in the ODM model because the concept of object identity states that every object has a unique identifier which is permanently associated with that object. In some models for secure ODM systems, polyinstantiation does not occur because of these globally unique object identifiers (Lunt and Millen, 1989). From a technical view point, this has advantages for preserving referential integrity; however, a problem still exists from the user's view point since the same real-world object is still represented by two objects. (Thuraisingham, 1989b)

SECURITY IN THE ODM STANDARDIZATION PROCESS

Integrating security into the standardization process for the ODM model has two aspects. One is the formulation of standards for a secure ODM model; the second is the addition of security features to the ODM model itself. Secure data models are the subject of much research. Several researchers have proposed secure relational data models, and several secure RDBMSs are available as products. However, open issues have been recognized and much research remains to be done. Standards for the relational data model itself have been available for several years (International Standards Organization, 1989): the standardization effort for the ODM model is not completed. Furthermore, the research reviewed in the previous section on secure ODM systems has only just begun. Clearly, it is not yet time for standardization of a secure ODM model: it is not too early, however, to add security features to the regular ODM model.

The authors of this paper take the position that security features should be part of the emerging standards for the ODM model. Experience with security for the relational data model has shown the

importance of considering security as an integral part of a reference model. Because this has not been done in the past, researchers working on MLS RDBMSs now find that their products cannot adhere to the proposed SQL2 standard for referential integrity. Even in standard RDBMSs, integrity constraints and secrecy constraints conflict, while such fundamental issues as the order of processing for integrity and secrecy constraints must still be considered (DBSSG, 1990).

The draft *Reference Model for Object Data Management* already includes a number of important security features, such as identification and authentication, audit, and a generalized notion of access control (authorization). The following are additional features suggested for inclusion in ODM model standards. These features support many of the concepts for information security identified in the second section of this paper, and are felt by the authors to be important to ordinary as well as secure ODM systems.

Access Controls. Support should be provided to the owners of data to permit selective access to other users based on semantically meaningful access controls. Such controls would include content-dependent access rules, which restrict access to an object based on its content , and context-dependent access rules, which restrict access to an object based on whether other objects are accessed in conjunction with the requested object.

Security Constraints. Security constraints include both secrecy constraints and integrity constraints. Secrecy constraints are predicates on the system that protect data from operations that are actually illegal. Integrity constraints, on the other hand, protect the system from operations that are invalid. Integrity constraints are already part of the draft ODM reference model. In a similar fashion, support should also be provided for system administrators to define secrecy constraints within the ODM system data language. Legal operations should be defined in terms of content, context, association, and aggregation constraints.

User Roles. The current list of roles of users of ODM systems should be expanded to include the notion of a System Security Officer for the administration of user accounts and audit data. Furthermore, support should be provided for basing system administration privileges on special *roles* that may be assigned to individual user accounts, instead of through the sharing of a single "super user" account as is frequently done in systems today. Examples of different system administration roles are operator, database administrator, and security officer. In addition, a role definition capability is needed for application-specific roles.

Object Boundaries. Although object boundaries are not security features, the determination of an object's boundary has important implication for the application of authorization controls and security constraints (Fernandez and Gudes, 1989), as well as for performance, access methods, and concurrency control (Dabrowski et al, 1990). Class hierarchies lead to a recursive definition of an object, and inter-object relationships potentially extend an object's scope even more. Therefore, the ODM model should provide a way of unambiguously determining the extent or scope of an object.

Finally, we would like to see a flexible ODM standard in order to avoid precluding assurance or other features necessary for a secure ODM model. For example, it may seem reasonable that discretionary access control for the the ODM model be done at the object level. However, a standard

requiring this type of access control would preclude using a message filter to mediate mandatory (and perhaps discretionary) access control in a secure ODM system. Similarly, some researchers (Thuraisingham, 1989a and b) feel that augmentation of an ODM system with logic programming is a better approach to constraints than methods. Examples of other ODM features for which rigid standardization has a potential security impact are concurrency control and clustering. Traditional locking techniques for concurrency control have covert channel problems associated with their use, while clustering can present problems in MLS systems because of issues concerning the storage of data at different classification levels in a single level file.

SUMMARY

We have looked at information security from the perspective of secrecy and integrity, and have defined a variety of security concepts that address those issues. Next, we have looked at the ODM model and identified how its features can support these concepts. Although acknowledging it is not yet time for a secure ODM model, we have taken the position that security features are important to any ODM system, and that many of these features are appropriate for inclusion in the emerging ODM model. These features address the areas of access controls, security constraints, user roles, and object boundaries. Decisions about the standardization of features affecting the ability of an ODM system to provide security should be flexible enough to allow security to be implemented where needed.

REFERENCES

Bell, D.and L. LaPadula, May 1973, *Secure Computer Systems: Mathematical Foundations and Model*, Technical Report M74-244, The MITRE Corporation, Bedford, MA.

Dabrowski, C. E., et al, April 1990, *Object Database Management Systems: Concepts and Features*, NIST Special Publication 500-179, National Institute of Standards and Technology, Gaithersburg, MD.

Database System Study Group, January 1990, *Proceedings of the Second Joint Meeting*, Orlando, FL.

Fernandez, E. B., and E. Gudes et al., May 1989, "A Security Model for Object-Oriented Databases," 1989 *IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA.

Gasser, M., 1988, *Building Secure Computer Systems*, Van Nostrand Reinhold Company Inc., New York.

Hinke, T. H. et al, October 1988, "A1 Secure DBMS Design," *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD.

International Standards Organization, November 1989, *Reference Model for Data Management DP10032*, ISO-IEC/JTC1/SC21/WG3/N1007.

Jajodia, S., and B. Kogan, May 1990, *Integrating an Object-Oriented Data Model with Multilevel Security*, Technical Report RADC-TR-90-91, Rome Air Development Center, Rome, NY.

Keefe, T. F. et al, October 1989, "SODA: A Secure Object-Oriented Database System," *Computers & Security*, Vol. 8, No. 6.

Lunt, T. F., "Multilevel Security for Object-Oriented Database Systems," *Proceedings of the Third IFIP Working Group 11.3 Workshop on Database Security*.

Lunt, T. F. et al, April 1988, "A Near-Term Design for the SeaView Multilevel Database System," *Proceedings of the 1988 IEEE Symposium on Security & Privacy*, Oakland, CA.

Lunt, T. F., and J. K. Millen, August 1989, *Secure Knowledge-Based Systems*, SRI-CSL-90-04, SRI International, Menlo Park, CA.

National Computer Security Center, 1985, *Trusted Computer System Evaluation Criteria*, Technical Report D0D 5200.28 STD, Ft. Meade, MD.

National Computer Security Center, 1989, *Trusted Database Interpretation (Draft)*, Ft. Meade, MD.

Object Oriented Database Task Group of the Database Systems Study Group, May 1990, *A Reference Model for Object Data Management (Draft)*, OODB 89-01R4, American National Standards Institute Accredited Standards Committee.

Smith, G. W., December 1988, "Identifying and Representing the Security Semantics of an Application," *Proceedings of the Fourth Aerospace Computer Security Applications Conference*, Orlando, FL.

Stachour, P. D., and B. Thuraisingham, June 1990, "Design of LDV: A Multilevel Secure Relational Database Management System," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 2, No. 2.

Thuraisingham, M. B., October 1989a, "A Multilevel Secure Object-Oriented Data Model," *12th National Computer Security Conference*, Baltimore, MD.

Thuraisingham, M. B., October 1989b, "Mandatory Security in Object-Oriented Database Systems," *Proceedings of the ACM Conference on Object-Oriented Programming Languages*, New Orleans, LA.

Thuraisingham, M. B., March/April 1990, "Security in Object-Oriented Database Systems," *Journal of Object-Oriented Programming*, Vol. 2, No. 6.

Thuraisingham, M. B., and F. Chase, October 1989, "An Object-Oriented Approach for Designing Secure Software Systems," *IEEE Cipher*, Fall.

A Strawman Reference Model in Transaction Processing for an Object Oriented Database

Chung C. Wang
Texas Instruments Incorporated *

Abstract

This position paper provides a strawman reference model which can be used to compare and reason about transaction management in an Object Oriented Data Base system (OODB). The model is described as consisting of a collection of characteristics that can be used for comparing existing and future features in transaction management of an OODB. Some of the features in this collection are really alternatives to one another. The purpose of inclusion of these alternatives is to help the evaluation process when developing standards.

1 Introduction

This position paper provides a strawman [1] reference model which can be used to compare and reason about transaction management in an Object Oriented Data Base system (OODBs). The transaction reference model that is orthogonal to an object data model deals with issues of concurrency control, database consistency, distributed processing, and transaction needs which are unique for an OODB application. The model is described as consisting of a collection of characteristics that can be used for comparing existing and future features in transaction management. Some of the features in this collection are really alternatives to one another. The purpose of inclusion of these alternatives is to help the evaluation process when developing standards.

The model is an English description of the design space of features of a domain and provides individuals with a way to discuss and compare the issues in the domain. Before describing the features of the model, we described some of the goals for the design of a reference model.

*Information Technologies Laboratory, Technical Report 90-08-xx, Computer Science Center, Texas Instruments Incorporated, P.O Box 655474, MS 238, Dallas, Texas 75265. Email: ccw@csc.ti.com Telephone: (214) 995-0337

[1] This paper is intended as a good start towards a transaction reference model; its treatment of transaction management is not intended to be complete. It is written in such a way that is consistent with OODBTG's OODB reference model. The transaction reference model is an orthogonal abstraction to the object data model.

2 Goals

Two different goals of a transaction module are to provide concurrency (high throughput volume, fast response time) and to maintain (at the same time) database consistency. Many application domains have different requirements on concurrency and require sometimes special mechanism and user interfaces for implementing transactions. The goal of standards is to achieve inter-operability of applications across softwares and hardwares of different vendors and to accommodate the needs of as many different applications as possible. This paper explores the issues when considering a reference model for the process of standardization in the transaction area of an OODB.

For the sake of discussion, we divide the features into basic and advanced features.

3 Basic Features

The basic features address the database consistency and concurrency control issues for "simple" transactions.

3.1 Database consistency

A transaction maps a *consistent* database state to a new consistent state *atomically*. It is a unit of *recovery*. The effect of a transaction on a database is either total or nothing at all. When using a persistent programming language, a programmer who is accustomed to a memory centered view must now use transactions to maintain database consistency. A challenging "seamless" problem in persistent programming language is to accommodate both the database centered view required by database consistency and the memory centered view accustomed to by a conventional programmer.

One solution is to treat cached (persistent) data in a user's memory space as a replicated partial copy of the data in a database. This cached data survives between transactions. The consistency problem of this cached data is basically the same as the primary copy in the database, but cached data plays a different role in how it is accessed and updated. A program should be able to read the cached persistent data outside a transaction and to take appropriate actions whenever it is modified by other transactions; it can be modified only *inside* a transaction using proper concurrency control tools. As an example, an object browser can manage its objects in a cache outside a transaction without blocking any other transactions from updating these objects.

3.2 Concurrency control

A correct and concurrent execution of transactions (with read and write conflict) is either *serializable* or *non-serializable*.

3.2.1 Serializable executions

This kind implements *interleaving* of transactions such that the execution is equivalent to some serial execution of the transactions in the classical read and write conflict. The basic modes are *pessimistic*, *optimistic*, and *semi-optimistic*.

In pessimistic mode, a transaction waits when there is a data conflict until the other transaction terminates. In optimistic mode, a transaction proceeds as if conflicts can never occur; the transaction aborts at commit time if database consistency is violated; otherwise the transaction commits. A transaction in optimistic mode relies on *validation* process to achieve serializability — not *wait*. In semi-optimistic mode, a transaction waits when there is a data conflict, but it *never* waits as long as in pessimistic mode. A transaction, in semi-optimistic mode, may receive information from another active transaction; it becomes therefore dependent on the other transaction and risks the probability of being aborted if the other one does.

Both the pessimistic mode and the semi-optimistic mode use locks for concurrency control and the optimistic mode, on the other hand, uses timestamps. The transaction executions in all three modes are serializable; the serialization orders for the pessimistic and semi-optimistic mode are determined at run time whereas the order for the optimistic is determined statically in transaction timestamp order.

All three modes will be discussed further in detail below. Since pessimistic concurrency control is the most widely used mode in commercial RDBMS, this mode is covered in more detail than the other two.

- **Pessimistic mode**

  Pessimistic mode is most suitable among transactions with high probability of data conflict. Pessimistic concurrency control places a transaction in a wait state when there is a conflict; the transaction proceeds when the conflict is resolved. A transaction, therefore, cannot introduce any inconsistency at the time of commit. Locking is a common way for implementing pessimistic concurrency control. Two phase locking protocol [1] is sufficient for ensuring serializability of an execution of transactions. A strict two-phase locking protocol, in which locks are not released until after the transaction commits/aborts, prevents cascade abort — a ripple effect on committed transactions that read the result produced prior to the time of commit/abort of a transaction. The following is a list of other locking related features:

  - **Lock types**

    The most commonly understood locks are *read* and *write* locks. A transaction waits when a lock request cannot be granted because of a conflict. In a persistent programming language, objects managed in a cache outside a transaction should not affect the serializability of transaction executions; these objects are not modifiable outside a transaction but must be synchronized whenever a newer version becomes available. An object in a cache *outside* a transaction can either have a *null lock* or *notify lock* [2].

    * *Read lock* is used when a transaction reads an object and wishes to be guaranteed such that no transaction can commit a newer version to the database while the lock is held. *Read lock* can be shared among transactions.

[2] An update transaction must not use *null lock* or *notify lock* in place of *read lock* for objects in its read set that can be modified by other transactions to cause non-serializable executions. Careful use of both *null lock* and *notify lock* should not affect the serializability of transaction executions. An alternative is to support *null lock* and *notify lock* inside a transaction and risks the probability of non-serializable execution of transactions.

* *Write lock* is used when a transaction wishes to write a new version of an object. *Write lock* is exclusive.
* *Null lock* is used for object managed in a cache outside a transaction. The locks on objects accessed inside a transaction can be turned into null locks after the transaction terminates.
* *Notify lock* is used for object managed in a cache outside a transaction. A program is interrupted whenever a new value is committed in the database copy of a notify-locked object. The locks on objects accessed inside a transaction can be turned into notify locks after the transaction terminates.

– **Implicit and explicit locking**

Implicit locking hides lock requests from a user and is therefore more seamless from a programming language user's point of view. But explicit locking offers more flexibility in concurrency control. For example, an application that wishes to prevent deadlock can explicitly lock all needed database objects at the beginning of a transaction.

– **Granularity**

Some possible choices of unit of locking in an implementation are a page, an object, a cluster of objects, an object graph, a partition of a database, a database, and a collection of databases. Using implicit locking, the source code can remain the same when running on systems assuming different lock granules, but the achievable degrees of concurrency are most likely system dependent. An application using explicit locking and accessing two objects at the same time in two processes will not run in a system that uses only page level locking and the two objects are from the same page.

– **Hierarchical locking**

When granules of locks are hierarchically structured, one needs to support hierarchical locking [2] to improve the level of concurrency. For example, a tree structured object graph may be *write-intent locked* and two leaf objects of the tree graph *write-locked* by different transactions. If hierarchical locking is supported, there should be first a standard way for specifying the hierarchical way of grouping of the objects and then a locking precedence between the different levels of a hierarchy.

– **Upgrading locks**

A *read lock* can be changed to a *write lock* (but not vice versa) without affecting the serializability of transaction executions. *Null lock* and *Notify lock* are for cached objects outside a transaction; these types of locks can be exchanged with each other without impacting the serializability of the transaction executions. The lock on an object that is *null locked* or *notify locked* outside a transaction must be "refetched" either with a *read* or a *write lock* when the object is needed inside a transaction.

– **Propagating locks**

An application either fetches an object with an explicit lock type or faults in an object with a default lock type. Should the lock type of a faulted object be the same as the object from which the faulted object is derived? An object graph, in general, contains several paths to an object; should an object with a *read lock* be upgraded to a *write lock* if it is reachable from an object on another path with a *write lock*?

– **Synchronous and asynchronous requests**

A lock request function is synchronous if the function returns only when the requested lock is granted and asynchronous if the function returns immediately with a status of the lock request. A fully general asynchronous feature has the possibilities of multiple outstanding lock requests from a single transaction and increases therefore the complexity of the deadlock detection algorithm. A reasonable compromise in the asynchronous feature is to return the function call immediately indicating that the lock request is ignored if the lock is unavailable; a transaction must re-issue the request at a later time if the needs remain. This compromised asynchronous

approach has no negative impact on two phase locking protocol.

- **Deadlock handling policies**

  Deadlock (and also livelock [3]) occurs in a DBMS. Techniques for solving them include *prevention*, *avoidance*, and *detection and resolution* schemes. Applications have different requirements. Some applications may tolerate deadlock detections and resolutions and others may not. For example, a computer integrated manufacture (CIM) application that controls a physical robot on the factory floor cannot have a transaction being aborted due to deadlock. Standards for OODB should offer a choice of policies in handling deadlock on a per transaction basis to an application.

- **Restart**

  From database consistency point of view, there is little problem in restarting a transaction. In persistent C++, a program using transaction to perform a task must be coded carefully if the transaction may be aborted by the system; there are transient data which have been altered by the aborted transaction. Restart of a transaction can be made invisible to an application program if transient data are also restored to their state at the beginning of the aborted transaction.

- **Optimistic mode**

  Optimistic mode is best suitable in application domains that have little data conflict. Optimistic concurrency control lets a transaction proceed as if there is no conflict and performs a validation check at the time of commit of a transaction. The system assigns a unique timestamp to a transaction when it first begins to run. The transaction, during its life time, has a consistent view of the database. At commit time, the system checks, based on the timestamp, whether this consistent view of the transaction has been changed during the interval between the transaction starts and ends; if this consistent view still holds, the transaction is committed; otherwise the transaction is aborted and restarted.

- **Semi-optimistic mode**

  The reason that two phase locking works in pessimistic concurrency control is that cycles are prevented from occurring in a transaction serialization graph [4, 5]. If a DBMS maintains a transaction serialization graph among transactions, it replaces the two-phase locking protocol with cycle detection algorithm, and it aborts transactions that are parts of or depend on a cycle of a transaction serialization graph [3]. Serializable transaction executions can again be achieved by letting transaction commit according to their dependency order kept in the transaction serialization graph. This serialization order is determined at run time. The mode of concurrency control is called *semi-optimistic*.

  Semi-optimistic concurrency control places a transaction in a wait state when there is a conflict, but the waiting period is not as long as required in a two phase locking protocol. A transaction in semi-optimistic mode does not have to hold on to a lock until it terminates as required by strict two phase locking protocol, and a lock on an object can be released as soon as the object is used or the modified object is saved in a database. A transaction may acquire a lock that is released by another currently active transaction and becomes a dependent of the other transaction. Being optimistic about the fact that the other transaction will eventually commit, the transaction risks the probability of being aborted.

  Conversely, a transaction has the option to prevent itself from being aborted due to dependency on other transactions by restricting itself to acquire only locks that have never been held by any currently active transactions; this transaction then behaves exactly as transactions in pessimistic mode.

  Problem of database inconsistency is detected much sooner in semi-optimistic mode than in optimistic mode. A transaction in semi-optimistic mode is aborted as soon as it becomes part of or depends on a detected cycle in an associated transaction serialization graph, whereas a transaction in optimistic mode is aborted no sooner than it is ready to commit.

A system that supports more than one mode (i.e. *mixed* [6] mode) at the same time has some transactions running in one mode and others in a dif-

[3] Serialization graphs and wait-for graphs are similar, but they serve different purposes: one for maintaining correctness and the other for detecting deadlock.

ferent mode. A transaction running in one mode should be able to run in a different mode at another time without recoding the body of the transaction. The inter-operability goal of standards is to allow a transaction that runs on one system supporting optimistic mode to run on another system supporting only pessimistic mode preferably with no recoding of the source.

Locks create a problem in a mixed mode system: they are used in pessimistic and semi-optimistic modes, but not in optimistic mode. Semi-optimistic mode is actually a superset of pessimistic mode. This difference on locking can be solved by either requiring that only *implicit* locking be used in pessimistic and semi-optimistic mode or ignoring all explicit lock requests when a transaction is running in optimistic mode.

3.2.2 Non-serializable executions

Concurrency control in most commercial DBMS is based on read and write conflict. Recent research, however, has been directed away from serializability with respect to read and write conflict. Spector's [7] semantic based concurrency control defines dependency relations among the operations of an abstract data type (ADT). The admissible executions are serializable with respect to the operations of an ADT, but *non-serializable* with respect to read and write. Korth [8] assumes that the database can be partitioned based on user supplied consistency predicates such that serializable execution of transactions are required only at the individual partition level; serializability of transactions at the database level is therefore not necessary in Korth's model. Garcia-Molina and Lynch [9, 10] let a transaction be broken into breakpoints of different granule sizes and allow transactions to be interleaved at different breakpoints according to their semantic classes. Avalon C++ [11] supports an *atomic* data type that guarantees not only persistence and recoverable, but also atomic update of an object. The *subatomic* data type of Avalon C++ supports conditional critical section handling and state information based concurrency control.

4 Advanced Features

An application often stretches the limitation of the basic features. The tools for supporting cooperative designs, long duration transactions, nested transactions, distributed transactions, transient data, multiple databases, logging and recovery are covered in the section as advanced features of an OODB.

4.1 Multi-threaded transaction and cooperative design

Multi-threaded transaction extends the capability of grouping operations into one atomic step from a single thread to multiple threads running on different workstations. A multi-threaded transaction supports cooperative design work by modeling each team member's work as a thread (either light or heavy weight process) and the entire team's work as a transaction. Members of a team, being threads of a transaction, have shared access to each other's persistent data — a necessity for cooperative design work. The entire team's work, as a transaction, preserves the consistency of the database.

The sharing of objects among threads of a transaction requires a different kind of concurrency control; execution of threads, in general, is non-serializable. Locking without two-phase protocol at the thread level suffices to solve concurrent access to the shared objects. If two-phase locking is the means to ensure serializability of executions at transaction level, it also applies to each multi-threaded transaction. An OODB should hide the difference between the thread level locking and the transaction level locking at user level as much as possible.

4.2 Nested transactions

A nested transaction [12, 13] is a way of grouping operations as an atomic step *within* a transaction. In particular, a sequence of code of a method in an object-oriented application can be grouped as a transaction and methods are often called in a nested manner. Consequently, transactions are nested naturally in an object oriented application.

Nested transactions also support intra-transaction concurrency. In a single threaded implementation of nested transactions, transactions are nested as stack operations; sibling transactions at the same level must be executed one at a time. On the other hand, in a multi-threaded implementation, sibling transactions at the same level can be run as separate processes. Clearly, an application that is coded for a system supporting mutli-threaded, nested transactions has to be recoded to be able to run on a system supporting only single-threaded, nested transactions.

4.3 Long duration transactions

Applications of an OODB often include long duration transactions. The problems that are amplified when the duration of a transaction is stretched into hours, days, or weeks are the following.

- A short transaction may be blocked by a long duration transaction for unacceptable length of time.
- The amount of work lost due to aborting a long duration transaction or system failure may be unacceptable.
- The lock table kept in a volatile memory has a greater probability of being destroyed due to a system failure during the life time of a long duration transaction.

The following are features related to handling long duration transactions.

4.3.1 Checkpoint

To avoid losing work due to an unwanted abort, a transaction can issue a checkpointing to save its partial results at appropriate intervals; a transaction restarts after an abort at the last checkpointed state of the transaction.

4.3.2 Piggy-backed transactions

An alternative way of supporting checkpointing is to introduce *piggy-backed transactions*. A long duration transaction is mapped to a sequence of piggy-backed transactions such that when one of the transactions commits, its successor transaction inherits all its locks and object cache.

4.3.3 Check-in-and-check-out model

Conceptually, a long duration transaction [14] can be implemented by checking out the data from a public database into a private database and merging the results back into the public database when the work is done. A long duration transaction holds the locks in a public database on the objects that are checked out to prevent their access by other transactions.

4.3.4 Persistent locks

Lock tables are usually kept in a main memory for minimizing the overhead in using locks, but locks for objects checked out for a long duration transaction must survive system crashes and therefore need to be stored in a database as persistent objects. Persistent lock tables are removed from the database when the objects are checked in. The active locks in the lock table are the union of those locks normally kept in a main memory and all persistent locks.

The features covered so far have solved the problem of work loss due to a system crash in running a long transaction, but they do not address fully the problem of concurrent access to shared data by a long transaction and other short transactions. The branching versions [15] model discussed next allows concurrent updates of different branching versions of the same objects by two or more (long or short) transactions.

4.3.5 Branching versions model

A branching version model for long transactions does not require maintaining two databases — public and private. A long duration transaction is modeled by a sequence of *regular short* transactions that operate on a private branch of versions of the database. Transactions accessing data of different branches of versions do not interfere with each other. Transactions that model parts of a long duration transaction save their results through regular commits; risks of of losing work in a long transaction are minimized.

The multiple branching versions for long duration transaction are orthogonal to both multi-threaded transactions and nested transactions; for example, an application contains a multi-threaded transaction that has nested subtransactions and operates on a private branch of versions.

4.4 Distributed transactions

A distributed database has disjoint data of the database stored at multiple sites in a network. A distributed transaction is a way of grouping a set of operations that accesses these data at multiple sites of a network in one atomic step. The standardization issues exist in (1) user interface for grouping the operations of a transaction at a site, (2) two phase commit protocol, (3) distributed deadlock handling, and (4) node and network failure handling.

A client-server model partitions a database application into two generic classes of processes: typically the server processes contain the database service related code, and the client processes contain application specific code. The most general configuration consists of multiple server machines and multiple

Parameter type	Possible Values
Concurrency control mode	PESSIMISTIC/OPTIMISTIC/SEMI-OPTIMISTIC
Read-only transaction	YES/NO
Lock request type	SYNCHRONOUS/ASYNCHRONOUS
Lock to be converted to at end of transaction	NULL LOCK / NOTIFY LOCK
Check-in/check-out	CHECK-IN/CHECK-OUT/NULL
Persistent locks	YES/NO
Locks and cache held for next transaction	YES/NO
Deadlock handling policy	PREVENTION/DETECTION
Branching version	<version-name>
Multi-threaded transaction	<transaction identifier>/NULL
Nested transaction	<parent transaction identifier>/NULL

Table 1: Transaction parameters.

client machines of a network supporting distributed, multi-threaded and nested transactions. Most vendors do not support this general configuration in their early releases. Intermediate configurations include (1) multiple user single machine, (2) one client machine and one server machine, (3) multiple client machines and one server machine, (4) one client machine and multiple server machines, and (5) multiple client machines and multiple server machines. One of the standardization issues is the visibility of the server sites to an application program.

4.5 Transaction on transient data

Transaction enables *concurrent* access of persistent data and supports *rollback* of changes on persistent data through transaction abort. In a persistent programming language, transaction can be an orthogonal property to cover both persistent and transient data. The *rollback* feature on transient data simplifies the implementation of what-if analysis. The *concurrent* access of shared transient data in a serializable way guarantees the consistency of the data. In an environment such as MACH that supports large virtual address space and light weight processes, transactions on transient data offer most of the benefits of a main memory database.

4.6 Multi-base transactions

In a client server model, a client application could access data from several servers running different vendors' OODBs. In addition to the consistency at individual database level, there is also the inter-database consistency problem when these databases are considered as a whole, as in distributed transactions. The standardization of the interface protocol between client and server should include a standard two-phase commit protocol to achieve inter-database consistency when multiple vendors' databases are accessed in one transaction.

4.7 Logging and recovery

The logging needs for recovery differ for transaction abort, machine and medium failures. Transaction abort is already covered briefly. Medium failures require that backup of the database be taken periodically. The logging requirement for machine failures depends very much on the scope of data saved in the database. For example, a database that contains versions of an object needs to log only the timestamps of committed transactions. On the other hand, an OODB that supports in-place updates needs *write-ahead logging* for failure recovery. If the recovery process requires *re-do* of some of the transactions, a *write-after log* must also be used.

5 Application program interface

The set of application program interfaces (API [16]) for a persistent programming language should ideally be as small as possible. A minimal set of API for transactions includes *transaction_begin*, *transaction_commit*, and *transaction_abort*. The parameters for *transaction_begin* should include the options available for running a transaction. Table 1 summarizes briefly the options described in the paper. The default values and the return values that are not described in Table 1 should also be specified in the standardization process. These parameters may also be specifiable in environment variables so

that the same source code can be run with different options or on different vendors' systems with minimum modification. The parameters of both *transaction_commit* and *transaction_abort* are empty. The return values of *transaction_commit* should cover the possible results of a commit operation.

6 Comparison with other OODBs

Table 2 and Table 3 present a comparison among a representative (but not complete) sample of OODBs in terms of the basic and advanced features of transaction they support. The systems included in this comparison are (in alphabetical order): Carnegie Mellon University's Avalon C++ [11], Servio Corporation's GemStone [17], Obectivity's Objectivity/DB [18], Object Design's ObjectStore [19], Ontologic's Ontos [20], Altair's O_2 [21], MCC's ORION [22], UC Berkeley's Postgres [23], Versant's VERSANT [24], and Texas Instruments' Zeitgeist [25]. Although our comparison has been based on published material publicly available to us and through private communications with some of the vendors as of this writing, we know the table may still not be accurate. We invite corrections.

In the table, N stands for No, Y stands for Yes, and ? stands for unknown or unclear to a system's transaction features. The rows in the table describe OODBs and columns describe concurrency control and transaction features. Table 2 describes the basic features and Table 3 describes the advanced features. These features are numbered in the table as follows.

1. Pessimistic mode
2. Optimistic mode
3. Semi-optimistic, mixed, and other modes
4. Read and write locks
5. Null lock
6. Notify lock
7. Other lock types
8. Page level lock
9. Object level lock
10. Cluster lock and others
11. Hierarchical locking
12. Explicit locking
13. Implicit locking
14. Synchronous lock request
15. Asynchronous lock request
16. Deadlock detection - timeout
17. Deadlock detection - wait for graph
18. Multi-threaded transaction
19. Nested transaction
20. Check-in/check-out model
21. Public and private database
22. Long transaction through versioning
23. Persistent locks
24. Multiple clients/servers
25. Transaction on transient data
26. Write-ahead/write-after logging
27. Backup and recovery

6.1 Notes on the Comparison

The following lists some additional comments on the features in Table 2 and Table 3.

1. A nested transaction in Avalon C++ can also be made a *top level* nested transaction.

2. GemStone's concurrency control is based on an optimistic foundation with pessimistic extensions.

3. Objectivity/DB supports locking at *container object*, *database object*, and *federated database* levels.

4. Ontos supports both pessimistic and optimistic mode of concurrency control. The optimistic mode in Ontos is not timestamp based. A transaction in optimistic mode does not wait for other transactions running in optimistic mode when there is a conflict, but it waits when there is a conflict with a transaction running in pessimistic mode.

5. Avalon C++, VERSANT, and Ontos, and Zeitgeist support multi-threaded transactions for cooperative design work.

6. Objectivity/DB, ObjectStore, and VERSANT support persistent locks for check-in and check-out model. (For different purposes, Postgres requires that locks on *procedure type* data to be persistent.)

7. Most vendors implement a long duration transaction as a set of short transactions that either operate on a private database or a private branch of versions of the database.

8. Postgres and Zeitgeist supports linear versions at the storage level.

System	CC Mode			Lock Types				Lock Granules				Lock Req. Types			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Avalon C++	Y	N	Y	Y	N	N	Y	?	Y	Y	N	Y	Y	Y	N
GemStone	Y	Y	Y	Y	N	N	Y	N	Y	N	N	Y	N	N	Y
Objectivity/DB	Y	N	N	Y	?	?	?	N	N	Y	N	Y	Y	Y	?
OjectStore	Y	N	N	Y	N	?	Y	Y	N	?	N	N	Y	Y	Y
Ontos	Y	N	Y	Y	Y	Y	Y	N	Y	Y	?	Y	Y	Y	Y
ORION	Y	N	N	Y	?	?	Y	?	Y	Y	Y	?	?	Y	?
O_2	Y	N	N	Y	?	?	Y	Y	N	N	N	Y	?	Y	Y
Postgres	Y	N	N	Y	N	N	Y	Y	Y	Y	?	?	Y	Y	?
VERSANT	Y	N	N	Y	?	Y	Y	N	Y	Y	Y	Y	Y	Y	Y
Zeitgeist	Y	N	Y	Y	Y	Y	Y	N	Y	Y	N	Y	Y	Y	Y

Table 2: Comparison among a representative sample of OODBs.

System	Transaction Related Features											
	16	17	18	19	20	21	22	23	24	25	26	27
Avalon C++	Y	Y	Y	Y	N	N	N	N	Y	N	Y	Y
GemStone	N	N	N	?	N	N	N	N	Y	N	Y	Y
Objectivity/DB	Y	?	?	?	Y	Y	Y	Y	Y	N	Y	Y
ObjectStore	?	Y	N	Y	Y	Y	Y	Y	Y	N	Y	Y
Ontos	?	Y	Y	Y	Y	?	Y	N	Y	N	Y	Y
ORION	?	?	N	Y	Y	Y	Y	?	Y	N	Y	Y
O_2	?	?	N	?	?	?	?	?	Y	N	Y	Y
Postgres	?	Y	N	?	N	N	N	Y	?	N	Y	Y
VERSANT	Y	Y	Y	?	Y	Y	Y	Y	Y	N	Y	Y
Zeitgeist	Y	Y	Y	Y	?	?	Y	N	Y	N	Y	Y

Table 3: Comparison among a representative sample of OODBs — continued.

9. Zeitgeist also supports branching versions at storage level. Different versions of an object have the same object identifier. Consequently, an object graph is stored independent of versions. Versions of different branches of an object are locked separately so that a long transaction accessing one branch of versions of an object does not interfere with transactions accessing a different branch of versions of the same object.

10. Some features marked Y in Table 2 and Table 3 may not be available in current releases.

7 Conclusion

The key issue in transaction processing is database consistency and performance. The differences in applications and their performance needs often require different kinds of concurrent control policies to be implemented in transaction modules. The paper has discussed some of the alternatives from which a reference model for standardization in the transaction area must choose and the issues from both the user's and implementor's view point.

References

[1] Eswaran, L., Gray, J., Lorie, R., and traiger, I., "The Notion of Consistency and Predicate Locks in a Database System," *Communication of ACM* 19(11), Nov., 1976.

[2] Gray, J., "Notes on Data Base Operating Systems," IBM, San Jose, Calif., RJ2188, 1978.

[3] Ullman, J., *Principles of Database Systems.* Computer Science Press, Rockville, MD, 1980.

[4] Papadimitriou, C., *The Theory of Database Concurrency Control.* Computer Science Press, Rockville, MD, 1986.

[5] Bernstein, P., Hadzilacos, V., and Goodman, N., *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA., 1987.

[6] Maier, J. and Stein, J., "Development and Implementation of an Object-Oriented DBMS," in B. Sriver and P. Wagner, eds *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, MA, 1987.

[7] Schwarz, P. and Spector, A., "Synchronizing Shared Abstract Types," *ACM Transactions on Computer Systems* 2(3), Aug., 1984.

[8] Korth, H. and Kim, W., "A Concurrency Control Scheme for CAD Transactions," University of Texas Computer Science Department Report TR-85-34, Dec., 1985.

[9] Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems, 8(2)*, June 1983.

[10] Lynch, N., "Multilevel Atomicity—A new Correctness Criterion for Database Concurrency Control," *ACM Transactions on Database Systems, 8(4)*, Dec., 1983.

[11] Eppinger, J. L., Mummert, L. B., and Spector, A. Z., "Guide to the Camelot Distributed Transaction Facility including the Avalon Language," Carnegie Mellon University, Pittsburgh, PA, 1989.

[12] Moss, J.E.B., "Nested Transactions: an Approach to Reliable Distributed Computing," MIT Technical Report MIT/LCS/TR-260, 1981.

[13] Lynch, N., Merritt, M., Weihl, W., and Fekete, A., "A Theory of Atomic Transactions," MIT Technical Report MIT/LCS/TM-362, 1988.

[14] Bancilhon, F., Kim, W., and Korth, H.F., "A Model for CAD Transactions," *Proceedings of 11th International Conference on VLDB, Stockholm*, Aug., 1985.

[15] Joseph, J., Shadowens, M., Chen, J., and Thompson, C., "Strawman Reference Model for Change Management of Objects," X3/SPARC/DBSSG OODBTG Task Group Workshop, June, 1990.

[16] Perez, E., "A Strawman Reference Model for an Application Program Interface to an Object-Oriented Database," X3/SPARC/DBSSG OODBTG Task Group Workshop, Oct., 1990.

[17] Servio Logic Corp., "GemStone Product References," Servio Logic Corp., Beaverton, OR, 97006, Sept., 1989.

[18] Objectivity, Inc., *Objectivity/DB System Overview*. Objectivity, Inc., Menlo Park, CA, 94025, Mar. 1990.

[19] Object Design., "An Introduction to ObjectStore, Release 1.0." Object Design, Burlington, MA, 01803, Mar. 1990.

[20] Andrews, T., Harris, C., and Duhl, J., "The Ontos Object Database," Ontologic Inc., Burlington, MA, 01803.

[21] Fernando V., Bernard, G., and darnis V., "The O_2 Object Manager: an Overview," *Proceedings of 15th International Conference on VLDB, Amsterdam*, Aug., 1989.

[22] Garza, J. F. and Kim, W., "Transaction management in an Object-Oriented Database System," Proceddings of the ACM SIGMOD 1988 Conference.

[23] Stonebraker, M., "The Design of the POSTGRES," in Proc. of the ACM SIGMOD 1986 Conference.

[24] Versant Object Technology Corp., "ObjectToday! Versant Object Technolgy's Quarterly Newsletter," Versant Object Technology Corp., Menlo Park, CA, 94025, June. 1990.

[25] Ford, S. et al, "ZEITGEIST: Database Support for Object-Oriented Programming," in *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems*, 1988.

Transactions and Versioning in an ODBMS

Katie Rotzell
Versant Object Technology
October 23, 1990

1. Introduction

Versant Object Technology builds and sells a distributed Object Database Management System (ODBMS), called VERSANT. From its founding (as OBJECT-Sciences Corporation), the company has believed in and addressed the importance of a complete object model, fully distributed data management, and flexible transaction management as fundamental components of any ODBMS. Several papers at the previous OODB Task Group Workshop addressed the need for a common transaction model and version support for interoperability of data managers and portability of applications. Versant supports the OODB Task Group's resolve to promote standards in these areas. This position paper proposes transaction model and versioning model requirements in support of the kinds of distributed data processing that ODBMS applications require.

2. Transaction Model

2.1. Goal

Many applications which are driving the demand for ODBMS, such as CAD and CASE tools, computer-aided publishing tools, and what-if analysis tools, were never able to realize productivity gains from earlier database technology for several reasons, one of them being the inappropriateness of the transaction model.

Conventional transaction models assume that a typical transaction is very short and accesses a small amount of data; if an application tries to lock an entity which is already locked by another application's transaction, it may wait for a certain amount of time (usually a few seconds) and then its transaction will be automatically rolled back by the system. Since the lifetime of a transaction is shorter than that of the application which begins it, in conventional transaction models there is no need to store locks persistently. We call these transactions "short transactions."

In contrast, a typical transaction found in CASE and the other areas mentioned above can last for days or longer and can span several applications. Such a transaction must have persistent locks and must present the application with multiple options as to how to proceed if a requested object cannot be locked. We call these transactions "long transactions."

Many applications whose data modeling and access needs suit ODBMS require long transactions. However, many of the commercial, data- processing applications, which previously seemed well-suited to earlier DBMS technology, will also benefit from the modeling and storage capabilities of ODBMS while still relying on a more conventional transaction model. The goal, then, is to combine the two models into a single unifying framework which meets the needs of both types of application. The VERSANT transaction model is architected as a single unifying model. The model consists of three main components: 1) a short transaction, which is the smallest unit of atomic update; 2) a session, which is the length of time an application is connected to the ODBMS; and 3) a long transaction, which is a unit of update of arbitrary length. In the VERSANT model, long transactions consist of sessions and sessions consist of short transactions, as Figure 1 illustrates.

2.2. Short Transactions

A short transaction is a conventional transaction such as those found in relational and other commercial DBMS's. A short transaction is an atomic unit of update; the ODBMS guarantees that either all of it is made permanent in the database, or none of it is, regardless of system crashes, network crashes, media failures, etc. A short transaction will very likely be distributed; that is, it may involve updates to databases that reside on different machines. Therefore, a distributed update protocol such as two phase

commit is a requirement. Within a short transaction, it is desirable to have the capability to set a savepoint and later be able to rollback to that savepoint. Short transactions provide locking ("short locks") and deadlock detection like conventional transaction models. Incomplete short transactions do not survive system crashes; if such a crash occurs, all the effects of any uncommitted transactions will be undone, including all short locks.

The following operations are required in a short transaction:

- Begin short transaction. Start a short transaction. Often this is done automatically, as the result of starting a session or ending the previous short transaction.
- Commit short transaction. Make all the changes visible in the database(s) and release all short locks acquired during the short transaction.
- Checkpoint-commit short transaction. Make all changes visible in the database(s) but do not release any locks, so the application can continue without having to re-acquire locks.
- Rollback short transaction. Undo all changes made during the short transaction and release any short locks that have been set.
- Savepoint short transaction. Mark all changes made to objects up to this point in the short transaction, for possible undoing back to this point.
- Undo savepoint. Undo all changes made during the short transaction since the savepoint was executed.
- Short-lock object for read. Place a short read lock on an object. Other transactions may read the object but not update it or delete it.
- Short-lock object for write. Place a short write lock on the object. Other transactions may not access the object.

2.3. Sessions

A session is the duration of an application's interaction with the ODBMS. When a session is begun, the ODBMS does initialization of system structures; when the session is ended, the ODBMS does summary clean-up. When a session is begun, a short transaction is automatically begun. A session takes place within a long transaction. Therefore, when a session begins, it can also begin a new long transaction or it can connect to an existing one. When a session ends, it can optionally end the long transaction it belongs to.

The following operations are required for a session:

- Begin session with new long transaction <name>. A new long transaction is begun. The long transaction is given a user- defined name so it can be identified when connecting subsequent sessions to it. If no transaction name is specified, the system assigns a default name.
- Begin session with connection to long transaction <name>. The session is attached to the long transaction identified, automatically participating in any long locks that were already set by the long transaction.
- End session. The long transaction (including all its long locks) remains in effect after the session ends.
- End session and commit long transaction. All changes made to all objects during the long transaction are made visible to users outside the long transaction. All long locks are released.
- End session and rollback long transaction. All outstanding changes made during the long transaction are backed out. All long locks are released.

2.4. Long Transactions

A long transaction is a transaction which spans multiple short transactions, multiple sessions, and multiple application processes, and whose lifetime does not have any system-imposed upper bound (unlike short transactions). A long transaction can be committed in its entirety, or rolled back.

The following operations are required for long transactions:

- <Begin long transaction>. See Sessions.
- <Commit long transaction>. See Sessions.
- <Rollback long transaction>. See Sessions.
- Long-lock object for read. A persistent lock is placed on the object. Since this lock is stored in the database, it survives system crashes. Any short transaction which belongs to this long transaction inherits this long lock. Other transactions may read this object, but may not update or delete it. In the case that the object is already locked with a conflicting lock (in this case, a write lock), proceed options allow the application to: wait for the lock; queue the lock request and continue; abandon the lock request and continue. The granularity of locking is at the object level rather than, for example, the page level. It is possible that more than one object is locked per request, depending on the type of object; for example, if the object to be locked is a container, all the objects it contains can be locked as part of the same request.
- Long-lock object for write. A persistent lock is placed on the object. Any short transaction which belongs to this long transaction may update the object. Other transactions may not read or write this object. As with long read locks, proceed options apply.
- Long-unlock object. Release the long lock on the object.
- Monitor object for: request pending, object updated, object deleted, object available. Notification is sent about this object when any of the specified events happens. Notification can go to the user or to the long transaction. Events include another transaction trying to lock the object; a transaction commits, changing the object; a transaction commits, deleting the object; the object is no longer locked. Users should be able to specify their own events as well.

2.5. Checkout/Checkin Between Group Databases and Personal Database

In any groupwork environment where multiple users are accessing shared databases, security and integrity concerns will make it necessary for concurrency control and recovery features to be running. However, if a user places a long lock on an object, it is most likely for his or her personal use. In this case, it is often desirable to move objects with long locks to a personal database where system features such as concurrency control and recovery features can be optionally turned off. The personal database can reside anywhere on the network, including on the local workstation, thus providing site autonomy.

An extension of the long lock mechanism in a long transaction provides this capability. The extension is called checkout/checkin, and it is analogous to checking a book out of and back into a library. When an object is checked out of a group database, a long lock is placed on it in the group database and a system copy (with the same object identifier) is placed in the personal database. Thereafter during the long transaction, any reading or writing of that object takes place in the personal database. When the object is checked in, it is written back to the group database from which it was checked out; its long lock is removed in the group database; and it is deleted from the personal database.

The operations required for checkout/checkin are:

- Begin session with new long transaction <trans_name> and personal database <db_name>. Associate this personal database with this long transaction, so that all checked-out objects have system copies placed in this database.
- Begin session with connection to long transaction <trans_name> and personal database <db_name>. Again, associate this personal database with this long transaction for all check-out objects.
- Checkout object for read. Long-lock the object for read, and place a system copy in the personal database for this long transaction.
- Checkout object for write. Long-lock the object for write, and place a system copy in the personal database for this long transaction.
- Checkin object. Long-unlock the object in the group database it belongs to, and remove the system copy from the personal database.

- Snapshot object. Create a new user copy of the object in the personal database for this long transaction. This is not a system copy; it is a replica of the original object but with a new object identifier. This command is used when it is necessary to have a copy of an object which is inaccessible due to a lock conflict, or for performing a dirty read of an object.
- <Commit long transaction>. Originally defined under Session. Appended here so that, during the commit, all objects checked out are automatically checked back in.
- <Rollback long transaction>. Originally defined under Session. Appended here so that, during the rollback, all objects checked out are automatically checked back in, with no changes.

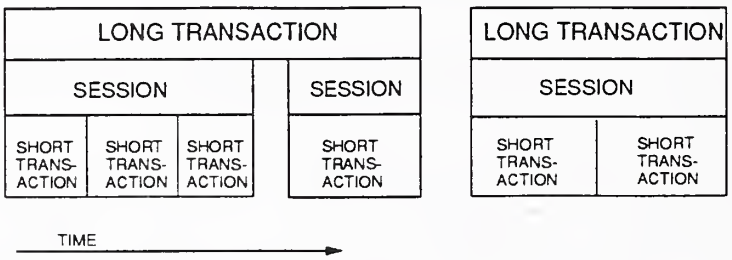


Figure 1.
Long Transactions, Sessions, and Short Transactions

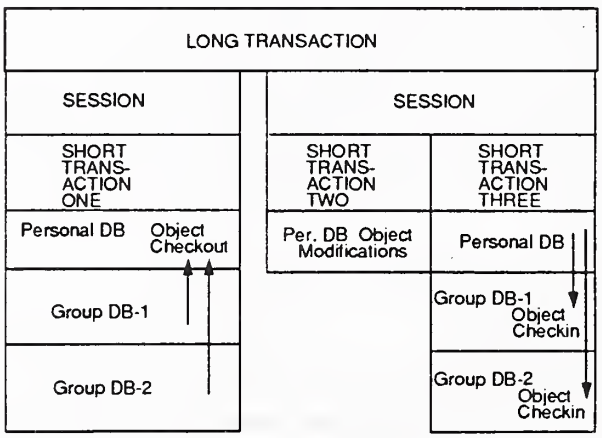


Figure 2.
Sample Long Transaction

2.6. Nested Transactions

The desirability of nested transactions as a general solution is still an open issue. However, this model can easily be extended to allow nesting of long transactions within other long transactions, with checkout of objects from a database associated with a "parent" long transaction to a database associated with a "child" long transaction.

3. A Version Model for Change Management

3.1. Goal

Versioning of objects and configurations is something that practically every ODBMS customer asks for. However, the specific requirements can vary remarkably from one customer to the next. Versant's position is to provide the minimum amount of versioning necessary for customers to tailor to their specific needs. Over time, organizations such as PDES (Product Data Exchange using STEP) and CFI (CAD Framework Initiative) will publish standard version models, which should be implementable in any ODBMS as a class library on top of the ODBMS primitives.

3.2. Requirements

The primitives provided by the ODBMS should be a minimal set. The ODBMS should support a version graph, where each version may have several successor versions (also known as alternatives), and may also have several predecessor versions (in which case it represents a merging of the predecessors). The ODBMS's responsibility is to manage the version graph; the ODBMS is not responsible for providing the merge or derivation process itself, since this is dependent on the class of object whose versions are being merged (e.g., you would not merge two versions of text and two versions of an image the same way).

In the proposed version model, each version is itself an object. Each version has a relationship to one or more other versions; possible relationships are "parent," "child," and "sibling."

Resolution of an object reference to a specific version must be done either statically or dynamically. This is because, in some cases, it will be desirable for a reference to an object to always return a specific version (as in a released design), while in other cases a reference should always return some default version (such as the latest). Of course, it should always be possible to get the default version from a specific version, and vice versa.

In the proposed version model, illustrated in Figure 3, there is a generic instance to which all dynamically bound references point. The generic instance acts as a forwarding mechanism, to forward the reference to whichever version is the default version at that time. Static references are created by pointing directly to the desired version.

The default version is the latest version "by default." However, it should be possible to set the default version to a specific version, or to the latest version that achieves a particular status. Versions can be designated a status such as: "transient" (in progress, highly unstable); "working" (unchanging, but may not be the released version); and "released" (final, frozen version). The default version should be settable to the latest of a given status. It must be possible to maintain versions across a distributed database, especially if versioned objects are to be checked out for update into personal databases.

3.3. Operations

The operations required for versioning are:

- Create versioned object of <object>. Creates a generic instance which points to <object> as its first and default version.
- Is-versioned <object>. Returns a boolean, indicating whether the object is a versioned object.
- Create new version of <version>. Creates a new object and adds it to the derivation graph for <version> as the latest version; links it to <version> as a child in the derivation graph.
- Get generic instance of <version>. Returns a handle to the generic instance for the derivation graph that <version> belongs to.

- Set status of <version> to {transient, working, released}. Sets the status of <version>.
- Set default version <generic instance> to {<version>, latest, latest released, latest working, latest transient}. Sets the version that dynamic references to the generic instance will return. The default version can be a specific version, or it can be the latest at any given time, or it can be the latest of a particular status at any given time.
- Get parents of <version>. Returns the versions that are parents of <version> in the derivation graph.
- Get children of <version>. Returns the versions that are children of <version> in the derivation graph.
- Add <version_1> to parents of <version_2>. Designates <version_2> a child of <version_1>, implying a merging of <version_1> with the other parents of <version_2>.

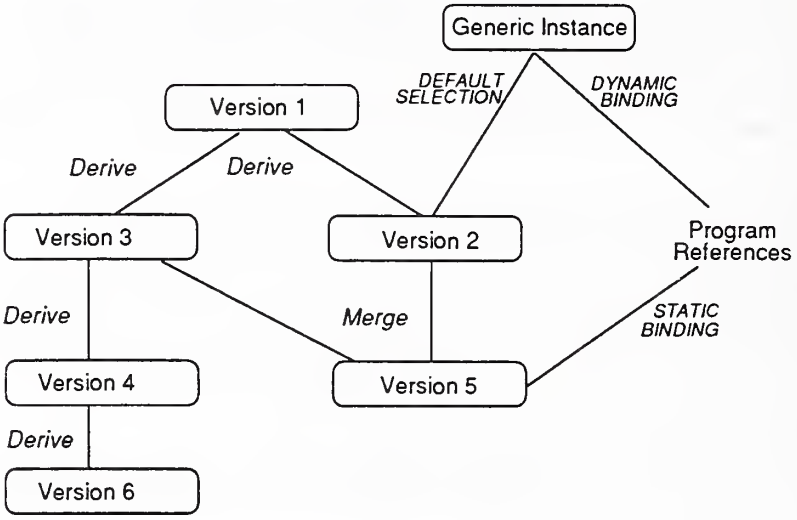


Figure 3.
Sample Version Graph

4. Implementation

A standard for transactions and versioning in ODBMS's needs to address not only the model but the interface as well, otherwise the goal of application portability cannot be reached. Various alternative implementations have been proposed for providing database functionality from an object programming language, including extending the programming language itself. However, making additional requirements on programming languages delays both the programming language standardization effort and the database standardization effort and risks adding unnecessary baggage to programming languages. Also, once you extend one language you have to extend every language, which adds even more confusion and delay to the process.

The Versant approach is to provide database functionality through class libraries. This does not get in the way of the language standards efforts, can be done across programming languages more easily, and is consistent with the object-oriented software development paradigm.

5. Acknowledgements

I extend my thanks to Dr. Hong-Tai Chou and Dr. Mary Loomis for reviewing earlier drafts of this paper.

Primitives for Schema Updates in an Object-Oriented Database System : A Proposal

Roberto Zicari

GIP Altaïr
78153 Le Chesnay Cedex, France
and
Politecnico di Milano,
Dipartimento di Elettronica
Piazza Leonardo da Vinci 32
Milano, Italy.

July 18, 1990

Abstract

Updating the schema is an important facility for object-oriented databases. However, updates should not result in inconsistencies either in the schema or in the database. We propose a classification of basic schema updates and define a set of parametrized primitives to perform schema updates which the designer will use to define his/her own update semantics.

1 Introduction

Schema evolution is a concern in object-oriented systems because the dynamic nature of typical OODB applications calls for frequent changes in the schema [Pan88]. We present a framework for schema updates for an object-oriented database system, and propose a set of parametrized primitives to perform schema updates which the designer can use to define his/her own update semantics. We will use in the rest of the paper the O_2 object-oriented database system [Bann88] [LRV88] as a representative example. However,the proposed language primitives can be easily modified to work for other object-oriented database management systems.

1.1 Preliminary O_2 concepts

In this Section we briefly recall the fundamental concepts of O_2 which are relevant for our discussion. The reader is referred to [LeRi88], [LeRi89a],and [LeRi89b] for a formal definition of the O_2 data model and to [Vel89] for the description of the system architecture. O_2 is an object-oriented database system and programming environment developed at Altaïr. Classically, in object-oriented data models, every piece of information is an object. In the O_2 data model, both *objects* and *values* are allowed. This means that, in the definition of an object, the component values of this object do not necessarily contain only objects, but also values. In O_2 we have two distinct notions: *classes* whose instances are objects and which encapsulate data and behavior, and *types* whose instances are values. To every class is associated a type, describing the structure of its instances. Classes are created using schema definition commands. Types are constructed recursively using *atomic types*

(such as integer, string, etc.), *class names*, and the *set*, *list*, and *tuple* constructors. Therefore types can be complex. Objects have a unique internal identifier and a value which is an instance of the type associated with the class. Objects are encapsulated, their values are not directly accessible and they are manipulated by *methods*. Method definition is done in two steps: First the user declares the method by giving its *signature*, that is, its name , the type of its arguments and the type of the result (if any). Then the code of the method is given. In O_2, the schema is a set of classes related by inheritance links and/or composition links. The inheritance mechanism of O_2 is based on the subtyping relationship, which is defined by a set inclusion semantics. Multiple inheritance is supported. O_2 offers a compile-time type-checker in an attempt to statically detect as many illegal manipulations as possible of objects and values. Objects are created using the "new" command. If a class is created "with extension" then a named set value is created which will contain every object of the class and will persist.

1.2 Updates: What do we want to achieve?

Updates can be performed both at schema and object level. An update is a dynamic modification. Changing the schema may logically imply changing all objects which are related to the portion of the schema which has been updated. Changing a specific (named) object does not imply a schema update, the update is limited to the object(s) specified and possibly to some other object(s) related to the object which has been modified. The main goal of this paper is to define a "reasonable" minimal set of primitives for updating an O_2 schema, and show the problems which need to be solved in order to obtain a usable schema update mechanism.

Informally, the problem with updates can be stated as follows: We want to change the structural and behavioral part of a set of classes (schema updates) and/or of a set of named objects (object updates) without resulting in run-time errors, "anomalous" behavior and any other kinds of uncontrollable situation. In particular, we want to assure that the semantics of updates are such that when a schema (or a named object) is modified, it is still a consistent schema (object). Consistency can be classified as follows:

a. *Structural consistency.* This refers to the static part of the database. Informally a schema is structurally consistent if the class structure is a direct acyclic graph (DAG), and if attribute and method name definitions, attribute and method scope rules, attribute types and method signatures are all compatible. An object is structurally consistent if its value is consistent with the type of the class it belongs to.

b. *Behavioral consistency.* This refers to the dynamic part of the database. Informally an object-oriented database is behaviorally consistent if each method respects its signature and its code does not result in run-time errors or unexpected results (cf.Sect.2).

A more precise definition of consistency will be given in Section 2.

We will consider "acceptable" only those updates that do not introduce structural inconsistency, while we will allow behavioral inconsistencies that do not result in run-time errors. Any kind of behavioral inconsistency that has been caused by an update will be reported to the user (designer).

1.3 Paper Organization

In the rest of the paper, we will concentrate our attention on the problem of schema updates. The problem of object updates will be addressed in a forthcoming paper.

The paper is organized as follows: Section 2 describes in some detail the distinction between structural and behavioral consistency for the O_2 object-oriented database system. The semantics

of schema update primitives will be that of transforming the schema from one state which is structurally and behaviorally consistent, to another state which is structurally and possibly behaviorally consistent. We will see that behavioral consistency is much harder to obtain than structural consistency. Therefore, we will accept some kinds of behavioral inconsistencies, i.e.those inconsistencies that do not produce run-time errors, and we will signal to the user (designer) all behavioral anomalies that may be induced by an update. The same Section also briefly describes how updates could be performed by invoking an interactive tool. Section 3 defines the set of primitives for updating the schema. We first present a full list of update primitives, and then show how this list can be reduced to a minimal set of basic updates, sufficient to implement all other updates. Moreover, we will classify updates in three categories. Each category is explained in detail in Sections 4,5,and 6. Section 4 describes schema updates which only modify methods. Section 5 describes schema updates which only modify the type associated to a class. Section 6 describes schema updates which modify a class as a whole entity. Problems arising from all such updates will be illustrated in the respective Sections. Section 7 compares our approach with related work. Finally in Section 8 we present the conclusions and state some open problems.

2 Ensuring Structural and Behavioral Consistency: A Schema Update Tool

In this Section, we discuss two basic types of consistency relevant to the O_2 system (but in general to every object-oriented database system) , namely *structural* and *behavioral* consistency.

Structural consistency refers to the static characteristics of the database.

We recall here some of the notions of O_2 [LRV88], which will help us to define the notion of a consistent schema.

We denote $T(C)$ the set of all types defined over a class C. T(C) includes atomic types, class names, tuple, set and list types.

Inheritance between classes defines a class hierarchy: A class hierarchy is composed of class names with types associated to them, and a subclass relationship. The subclass relationship describes the inheritance properties between classes.

Definition 1 A *class hierarchy* is a triple (C, σ , $\prec$) where C is a finite set of class names, σ is a mapping from C to T(C), i.e. σ (C) is the structure of the class of name C, and $\prec$ is a strict partial ordering among C.

The semantics of inheritance is based on the notion of subtyping. The subtyping relationship $\leq$ is derived from the subclass relationship as follows:

Definition 2 Let (C, σ, $\prec$) be a class hierarchy, the subtyping relationship $\leq$ on $T(C)$ is the smallest partial ordering which satisfies the following axioms:

1. $\vdash$ c $\leq$ c', for all c, c' in C such that c $\prec$ c'. That is, a subclass is a subtype.

2. $\vdash : [\ a_1\text{: } t_1,...,a_n\text{: } t_n,...,a_{n+p} : t_{n+p}] \leq [a_1\text{: } s_1,...,a_n\text{: } s_n\ ]$, for all types t_i and s_i, i=1,...,n such that $t_i \leq s_i$. This is subtyping between tuple types. We can refine tuples by refining some attributes or by adding new ones.

3. $\vdash \{s\} \leq \{t\}$, for all types s and t such that $s \leq t$. This is subtyping between set types.

4. $\vdash < s > \leq < t >$, for all types s and t such that $s \leq t$. This is subtyping between list types.

5. t $\leq$ *any*, for all types t. The symbol *any* is a type by definition.

As inheritance is user given, some class hierarchies can be meaningless.

In a class hierarchy an instance of a class is also an instance of its superclasses (if any). Therefore, if class c' is a superclass of class c, then we must have that the type of c is a subtype of the type of c'. More formally:

Definition 3 A class hierarchy (C, σ, $\prec$) is *consistent* iff for all classes c and c', if c $\prec$ c' then σ (c) $\leq \sigma$ (c').

Example: This is a consistent class hierarchy. Class Employee is a subclass of Person, (i.e. Employee $\prec$ Person):

```
class Person
type tuple [name:string,
            age: integer,
            address: tuple [location: City,
                            street: string] ]

class Employee
type tuple [name:string,
            age: integer,
            address: tuple [location:City,
                            street: string],
            profession: string,
            company: string ]
```

A schema is also constituted of methods attached to classes. Methods have signatures.

Definition 4 A method *signature* in class C is an expression m: c x t_1 x ...x $t_n \rightarrow$ t, where m is the name of the method, and c, t_1 ... t_n are types. The first type c must be a class name and is called the receiver class of the method.

We are ready to define a schema.

Definition 5 An O_2 *database schema* is a 5-tuple S=(C, σ, $\prec$,M,N), where:

- (C, σ, $\prec$) is a consistent class hierarchy (see def.3)
- M is a set of method signatures in C
- N is a set of names with a type associated to each name

A schema is therefore composed of classes related by inheritance which follow the type compatibility rules of subtyping and a set of methods. Attributes and methods are identified by name. Within the schema, type attributes and method names have a *scope rule* (see def. 7). When we do not want to distinguish between a type attribute and a method name, we simply use the term *property*.

Now we are ready to define what we mean with structural consistency for a database schema.

Definition 6 A database schema S is *structurally consistent* iff it satisfies the following properties:

- if c $\prec$ c' and the method m is defined in c with signature m:c x $t_2 ... t_n \rightarrow$ t, and method m' is defined in c' , and m and m' have the same name, with signature m': c' x $t'_2 ... t'_2 \rightarrow$ t' , then $t_i \leq t'_i$ and t $\leq$ t' (covariant condition)
- the class hierarchy is a DAG

- if there are classes c1 and c2 having a common subclass c4, with a property name p defined in both c1 and c2, but not in C4, then there is another subclass c3 of c1 and c2 in which the property p is also defined and c4 is a subclass of c3.

The first property assures that method *overloading* is done with *compatible* signatures, the second property constrains the structure of the class hierarchy, and finally the last property eliminates multiple inheritance conflicts (also denoted as *name conflicts*). Definition 6 is important, because we will always consider schemes which are structurally consistent. An update to a schema is a mapping which transforms a schema S into a (possibly) different schema S'. Schemas S and S' have to be structurally consistent. The semantics of the schema update primitives will have to ensure *at least* that structurally consistent schemas are produced as a result of an update. In our approach name or type conflicts occurring as a consequence of an update will not be solved automatically by the system.

Behavioral consistency refers to the dynamic part of the database. It ensures that methods do indeed perform the "desired" task. As we have seen, the signature of a method is used to type-check the compatibility of the method in the class structure. This kind of check is therefore part of the process to ensure structural compatibility. However, checking the signature of a method is not sufficient to ensure behavioral consistency of the method. Updating a schema could result in anomalous behavior of some methods due to the *method dependency* (i.e. methods in their body may refer to other methods or public types), and in some subtle cases to the change of signature of methods as resulting from a change of the class hierarchy. Behavioral inconsistencies do not always result in run-time errors. There may be methods that do not fail as a consequence of an update, but may behave in a "modified" way after an update. The two notions of *method failure* (i.e. run-time errors) and *method's change of behavior* (i.e. the expected method's result is different) are conceptually different. In the rest of the paper, we will not make this distinction and we will only use the general term *behavioral inconsistency* . We have taken the approach of allowing transformations of schema which may lead to behavioral inconsistencies that do not result in run-time errors (updates that cause run-time errors are refused), and signaling to the designer all methods which may potentially be affected by an update, leaving to the user the responsibility for actions to be taken in consequence of these warnings. The list of available O_2 schema updates is given in Section 3. For such updates, detailed examples of consistency problems can be found in Sections [Zic89]. A detailed analysis of the issue of structural consistency is reported in [DeZi89]. This paper concentrates mainly on the definition of a set of basic schema update primitives.

2.1 The ICC: A Basic Schema Update Tool

The way the designer updates the schema should be a dialogue with an interactive tool called the Interactive Consistency Checker (ICC). The ICC is a basic update tool which, given a schema and a proposed update, detects whether inconsistencies may occur. It then refuses those updates which produce structural inconsistencies or behavioral inconsistencies which may imply a failure of some methods(i.e. run-time errors). The consistency check is performed in two steps: first the structural check, and then the behavioral check. If structural inconsistencies arise then the behavioral check is not performed and the update is refused. The reason for the refusal of the update is always given to the user.

3 Schema Updates

We present in this Section a complete list of basic updates one may want to perform on an O_2 schema. Updates are classified in three categories: Updates to the type structure of a class, to

methods of a class, and to the class as a whole. This classification is fairly similar to the one of [Ba87a,Ba87b]. However, the semantics of some updates is different. The main differences between the schema update characteristics of O_2 and other relevant systems will be discussed in Section 8.

SCHEMA UPDATES:

1. *Changes to the type structure of a class*

   Because in O_2 types can be arbitrarily complex, we have different ways to modify a class type. We can think of an update u which modifies the type structure T of a class C, as a mapping between types, $u : T \rightarrow T'$. Updates of this kind can be broadly classified in two categories: those for which $T' \leq T$ (we call them type-preserving), and those for which $T' \not\leq T$ (we call them non type-preserving). Of all possible type updates we list here only the most elementary ones:

   1.1 Add an attribute to a class type

   1.2 Drop an existing attribute from a class type

   1.3 Change the name of an attribute of a class type

   1.4 Change the type of an attribute of a class type

   Updates 1.1 and 1.3 are type-preserving, while updates 1.2, and 1.4 are non type-preserving.

2. *Changes to the methods of a class*

   2.1 Add a new method

   2.2 Drop an existing method

   2.3 Change the name of a method

   2.4 Change the signature of a method (this update may be also implied by a change to the class structure graph as defined below)

   2.5 Change the code of a method.

3. *Changes to the class structure graph*

   3.1 Add a new class

   3.2 Drop an existing class

   3.3 Change the name of a class

   3.4 Make a class S a superclass (subclass) of a class C

   3.5 Remove a class S from the superclass (subclass) list of C

A set of more involved changes to a schema (defined as a tree) can be found in [AbHu88].

The list of updates defined above can be reduced: There exists a basic set of updates which can be used to execute all other updates.

The basic set of updates is the following:

BASIC SCHEMA UPDATES:

1.1 Add an attribute to a class type

1.2 Drop an attribute from a class type

2.1 Add a method

2.2 Drop a method

3.1 Add a class

3.2 Drop a class

3.3 Change the name of a class

3.4 Make a class a superclass (subclass) of C

3.5 Remove a class from the superclass (subclass) list of C

The other updates in the previous list can be executed using sequences of basic updates.

```
(e.g.  1.3 = <1.2 , 1.1>, this equivalence does not hold at
                          instance level
       1.4 = <1.2 , 1.1>, this equivalence does not hold at
                          instance level
       2.3 = <2.2 , 2.1>
       2.4 = <2.2 , 2.1>
       2.5 = <2.2 , 2.1)
       3.3 = <3.2,  3.1>, this equivalence does not hold at
                          instance level)
```

The sequence of basic updates corresponding to a non elementary update has to be atomic, to avoid inconsistency.

It is worth noticing how the notion of *completeness* of a set of basic updates at *schema level* [Ba87a], that is whether the set of basic updates subsumes every possible type of schema change, is not necessarily the same one at the *database instance level* [Mai89].

4 Method Updates

In the rest of this section we will only consider the two basic method updates as defined in Section 2, that is:

- Add a method m in class C
- Drop a method m in class C.

The other updates to methods can be executed by using the two basic ones. We consider the two updates separately.

4.1 Adding a method in a class

We have a schema, and we want to add a method to a class of this schema. The syntax of the update is as follows:

```
add_method <m> in <C> (<signature>, <body>) || (from <C'>)
```

Square brackets indicates an optional parameter. The notation (p1)|| (p2) is used to denote two alternative parameters, p1 or p2.

Where:

< *m* >: is the method name to be added.

$< C >$: is the class where the method is added.

$<$ *signature* $>, <$ *body* $>$: this parameter (if specified) defines the method signature and the body of method m.

from $< C' >$: this parameter (if specified) indicates that the signature and body of the new method are those of a method with same name m defined in a class C', with $C' \neq C$.

Structural consistency of the schema implies that, when adding a method, we must ensure that no problems arise in (Def.6):

a) class structure

b) methods' name resolution

c) methods' signature compatibility

Addition of a new method m in class C modifies the scope of all methods with the same name defined in the superclasses of C. The system must automatically ensure such modifications. The scope of the newly added method is the standard one (cf.Section 2).

Introduction of a new method may result in a name conflict in the same class C (i.e. a method with same name was already defined in C) or in a subclass C' of C in case of multiple inheritance with other classes. In either case, the update is refused.

Insertion of a new method implies a static check of its signature. The signature of the new method should be type compatible with the signature of existing methods with the same name and related to it by class inheritance. An incompatible signature will imply the refusal of the update. The detailed specifications of the algorithms for preserving structural schema consistency are given in [DelC89].

Addition of a method to a class does not require to logically update the class extension.

The addition of a method to a class may lead to behavioral inconsistencies. Checking behavioral inconsistencies must be done by looking at the code of methods. In particular, we have inconsistencies if:

- the code of the new method M contains references to methods or classes that do not exist in the schema. In this case the update is refused, and the designer has to first define those methods and classes which are not already created.

- other methods may be affected by the introduction of the new method because in their code they use a method with the same name but different signature.

Because of the method calling dependency, other methods may be affected as well by the update. For this purpose a ***method dependency graph*** can be extracted by looking at the code of each method. The vertices of the method dependency graph are the method names and there is an edge from m to n iff m occurs in the code of n. In order to ease the task of building the method dependency graph, and also to reduce the search for dependent methods (see also updates to the class type, cf. Section 5), the system associates:

- to each class an import list which tells whether the class is using methods of other classes, and also some information which tells what other classes (their public types to be precise) are accessed without using encapsulation by methods of the class;

- to each method, the list of methods which are called outside the class where the method is defined, and which types methods access without encapsulation.

We will call this information *dependency information*.

Obviously, before inserting a method, besides checking its signature, its body has also to be analyzed and type-checked in order to avoid run-time errors. Now, the body of a method can be rather complex; it is therefore important to be able to extract the relevant information when type-checking the method code. When an update is performed this information can be used to decide whether the method still works properly or whether the method needs to be re-compiled, or whether in any case the method will not work any more (cf. 4.2).

4.2 Dropping a method from a class

The syntax of the update to delete a method in a class is as follows:

```
delete_method <m> in <C>
```

where: $< m >$ is the name of the method to be deleted. $< C >$ is the class where m is defined. *Constraints*: It is not possible to delete a method which is inherited from another class. Method m must be locally defined in C.

Deletion of a method m also implies the deletion of all references to m using the *m from C'* clause.

When removing a method from a class (as for all updates) we should assure the structural consistency of the schema and detect possible behavioral inconsistencies.

We discuss here only briefly the issue of structural consistency. The reader is referred to [DeZi89],[DelC89] for a detailed analysis of the issue of structural consistency.

Deleting a method m from a class C can cause name conflicts and signature incompatibility if C inherits from more than one class. This is due to the change of method scopes after the update. Detection of structural inconsistencies results in the refusal of the update. We are currently designing a tool to help the designer finding method dependency, in this case method references using the from clause. If we consider an intelligent advisor to aid schema updates then we could have a different solution as will be reported in Section 9.

Deleting a method does not imply a logical update to the class's objects.

Behavioral problems can be detected by checking the signature of a method, looking at the method dependency graph, and most importantly looking at the code of the method. Some kinds of behavioral problems are obvious, such as a reference to a method which has been deleted (and not replaced through the inheritance mechanism by another method), and are solved analyzing the method dependency graph. There however are other cases of inconsistencies that are not so immediate. The reader interested in this problem is referred to [Zic89].

5 Type Updates

This section describes updates which modify the type associated to a class. We restrict our attention to the basic updates:

- delete an attribute from the type associated to a class,
- add an attribute to the type associated to a class.

In fact, the results we present for these two updates also characterize the larger set of other possible O_2 class type updates.

In the rest of this section we discuss the two issues of structural and behavioral consistency separately.

5.1 Structural Consistency.

The type changes proposed do not affect the class DAG. A structural check must be performed to detect whether the new class type definition does not result in [DeZi89] :

- Name conflicts. E.g. the same attribute name is defined in two or more superclasses. This problem is similar to the one of updating a method (cf. Section 4).
- Type incompatibility in the inheritence hierarchy.

For such cases the update is refused.

When performing an addition(deletion) of an attribute (and in general for any change in the class type) it is also necessary to logically update the class extensions. The objects belonging to the class extension need to conform to the new class type definition. Default values can be assigned to the new attributes; while the values which correspond to the attributes removed in the schema must be deleted. As it is shown in [Zic89], a logical update to a class extension does not necessarily corresponds to a physical update in the database.

5.2 Behavioral Consistency.

In [Zic89] two methods to ensure behavioral consistency when updating a class type are presented. Informally , when updatinga class type, it is desirable to know which methods use the updated type and what is the method dependency graph.

6 Class Updates

Updates to a class as a whole entity are equivalent to the operations of manipulation of a graph (i.e. the class DAG). In fact, we have:

- add a node is equivalent to add a class;
- delete a node is equivalent to drop a class
- add an edge is equivalent to make a class a superclass (subclass) of another class;
- remove an edge is equivalent to remove a class from the list of superclasses (subclasses) of another class.

There are different ways to give the semantics to these updates. We provide a set of *parameterized* primitives to perform class updates which the designer will use to define his/her own update semantics. In particular, the updates which may have different semantic interpretations are the following:

- deletion of an edge
- addition of a node
- deletion of a class.

For such updates, the problems are:

- where to connect a class if it becomes disconnected from the DAG after the update?
- do the types and methods of the subclasses change after deletion of a superclass or removal of a class from their superclass list?
- where to place a newly created class in the DAG?

6.1 Addition of an edge

Adding an edge in the class structure corresponds to making a class C a superclass (or a subclass) of another class S. A structural check has to be done in order to ensure that no name conflicts or type-incompatibilty arise. Adding an edge modifies the scope of attribute and method names in the inheritance hierarchy. The update must not introduce cycles in the DAG.

The update has logically an impact on the class extensions because the attributes of C have to be added to all instances of subclasses of C (if any).

The syntax of the update is as follows:

```
add_edge <S-C>,
```

where S and C are classes already existing in the DAG. Class C is made a direct subclass of class S.

The addition of an edge may create behavioral problems in the same way as when adding a method or an attribute to a class.

6.2 Removal of an edge

Removing an edge may be obtained in different ways. Suppose we have an inheritance hierarchy composed of three classes Person, PhD, and Employee (with associated types Ta,Tc, and Tb) as follows:

```
Person   type Ta
|
PhD      type Tc
|
Employee type Tb
```

We want to perform the update: remove class PhD from the superclass list of Employee. This update is equivalent to saying: remove edge $< PhD - Employee >$. For this particular schema, the effect of this update is in disconnecting the class Employee from the DAG ; in fact PhD is the only superclass of Employee. To preserve class consistency ,the class Employee has to be connected to some other class(es) in the DAG. There are two possibilities:

- Class Employee is made a direct subclass of all direct superclasses of PhD; class Person in the example, or
- Class Employee is made a direct subclass of the system class OBJECT.

Both the interpretations are "reasonable" depending on what the user wants to do with the class Employee.

Moreover, when deleting the edge PhD-Employee we also must define what happens to the attributes of the type Tb (associated to Employee) which are inherited from the class PhD and not redefined in Employee. There there are two possibilities:

- class Employee loses all attributes inherited from PhD;
- class Employee does not lose attributes inherited from PhD; attributes which were inherited become locally defined in Employee.

Again, the two interpretations are both reasonable. The same considerations hold for methods as well. The use of a parameterized update operator allows the definition of different update strategies. The syntax of the delete edge primitive is as follows:

```
drop_edge  <S-C> [ [connect <C>  to  <class_list>],
                   [type and methods_of  <C> are not preserved]
```

where S is a superclass of class C. Square brackets indicate an optional parameter.

The semantics of the update is as follows:

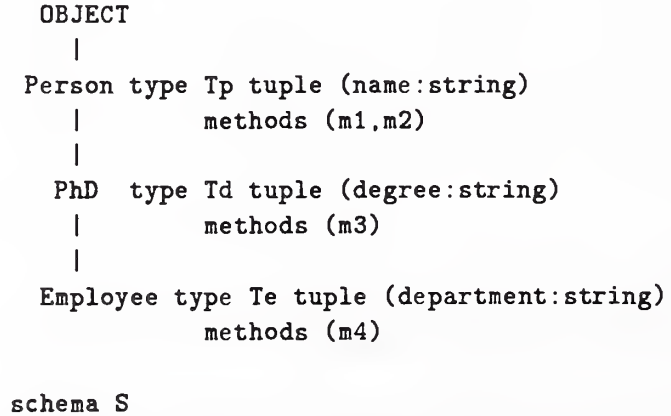
$< S - C >$: is the edge to be removed. Edge $< S - C >$ is removed from the class DAG.

connect $< C >$ to $< class - list >$: this parameter(if specified), indicates that in case class C becomes disconnected from the class DAG after deletion of the edge (i.e. S is the only superclass of C) then class C is made a direct subclass of the classes listed in $< class - list >$. Note that classes in the $< class - list >$ do not necessarily have to be superclasses of S. The update is refused if name conflicts or type-incompatibility arise when trying to connect class C. If the parameter is not specified then class C is made a direct subclass of the system class OBJECT. In this case, neither type-incompatibilty nor name conflicts occur.

type and methods-of $< C >$ are not preserved: this parameter (if specified), indicates that both the attributes of the type of class C and the methods inherited from class S and not redefined in C are deleted , this also implies the deletion in the subclass of C (if any) of all methods and type attributes that were inherited from the disconnected superclass and not locally redefined in the subclass. If the parameter is not specified then the type of C is unchanged (i.e. attributes which were inherited now become locally defined in C) and also methods of C are unchanged. Deletion of attributes in C may lead to name conflicts and type-incompatibility; in this case the update is refused.

We consider a simple example to show how this parameterized update models several "useful" situations.

Example: Consider the schema S composed of the inheritance hierarchy of classes Person, PhD, and Employee, as shown below :

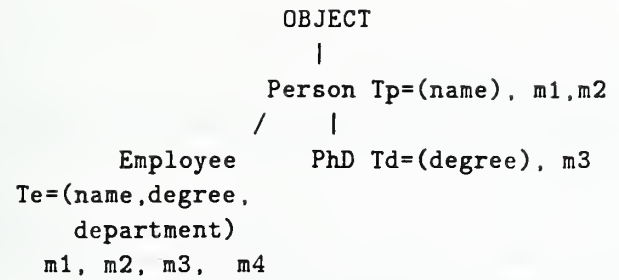


where class Person is of type Tp and has methods m1, m2. Class PhD inherits in its type Td the attribute "name" from Tp and has the attribute "degree" locally defined. PhD has a local method m3. Class Employee inherits the attributes "name" and "degree" respectively from Person and PhD, and has a local method m4.

Consider the following update:

```
a)  drop_edge <PhD-Employee> connect <Employee> to <Person> (S)
```

this update results in the following schema:



Class Employee is now a subclass of Person. Its type and methods have not been changed. Methods m1,m2 and m3 are now locally defined in Employee. Method m4 which was locally defined in Employee before the update is also unchanged. Note that after the update, all attributes of the type Te are *locally defined.*

It is also possible to specify that the attribute "name" of Te be the one inherited from Person. This can be done with either one of the following two updates:

```
<(delete_attribute name in Employee),(add_attribute name in Employee
  from Person)>
```

or in this case with the equivalent:

```
 delete_attribute name in Employee
```

In fact, in the example it is not necessary to add the attribute because class Employee does not inherits from multiple superclasses, and therefore no name conflicts occur in Employee.

Consider this other update on the previous schema S:

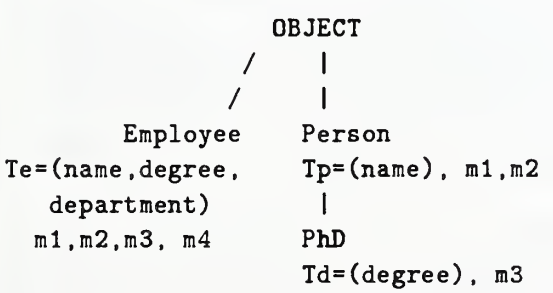
```
b) drop_edge <PhD-Employee> connect <Employee> to <Person>,
   type and methods_of <Employee> are not preserved (S).
```

This update results in the same schema as the previous update , but with the difference that the type Te does not have the attribute degree any more , that is Te=(department) and attribute "name" is inherited from class Person. Class Employee has method m4 locally defined, does not have method m3 any more and inherits m1 and m2 from Person.

Consider this update:

```
c) drop_edge <PhD-Employee> (S)
```

It results in the schema:



where class Employee is a direct subclass of class OBJECT. Its type and methods are unchanged.

If we consider the update:

```
d)  drop_edge <PhD-Employee>
    type and methods_of <Employee> are not  preserved (S)
```

We have as a result the same schema as the previous update, with the difference that class Employee does not inherit the attributes "name" and "degree" any more, that is Te=(department) , and has lost methods m1,m2 and m3.

As these examples show , the use of a parameterized operator allows the expression of several useful and consistent updates. The update has logically an impact on the class extensions if the class type C has to be changed, i.e. deletion of attributes in the class type of C.

6.3 Addition of a node

Adding a node corresponds to adding a new class in the class DAG. We allow addition of a class *in any position* in the class DAG. To preserve DAG consistency, when no position in the DAG is specified the new class is by default made a direct subclass of the class OBJECT. Adding a class in a different position in the class hierarchy implies adding an edge to a superclass S and/or an edge to a subclass C. The semantics of the update is the following:

a. The new class must not have a name which is already used for another class in the schema;

b. The new class becomes by default a direct subclass of the class OBJECT unless a different position is specified;

c. When adding a class C in an inheritance hierarchy no automatic attribute propagation is performed. The type of C must be completely redefined in order to be type-compatible with the types of the superclasses and subclasses of class C. If redefinition of attributes is not desired the update has an option which allows it to refer to attributes defined in superclasses of C.

The syntax of the update is the following:

```
add_class <C> [connect to <superclass Su> [ before <subclass Sb>] ]
type is <attribute definition> [from <class name>]
```

Brackets indicate an optional parameter.

We have:

- $< C >$ is the new class;
- connect to $<$ *superclass* $>$ [before $<$ *subclass* $>$] : indicates the position in the hierarchy where the new class has to be placed. Sb is a direct subclass of Su. If the subclass Sb is not indicated then the new class is a leaf of the DAG. Note that only one superclass and one subclass can be specified in the update. If class C has to be connected to more than one superclass or subclass, the explicit *add-edge* update has to be used. If the connect parameter is not specified, then class C is connected directly to the class OBJECT.
- type is $<$ *definition* $>$ (*from* $<$ *class* $>$): defines the type associated to the class. If the parameter (from) is used to define an attribute of the type then it refers to an existing attribute defined in the class S or in a superclass of S (if any).

Example: Consider the following schema:

```
OBJECT
  |
Person  T1=(name)
  |
Employee T3=(name,degree,department)
```

The following update:

```
add_class <PhD> connect to <Person> before <Employee>
    type is tuple (name:string,degree:string)
```

results in the schema:

```
 OBJECT
   |
Person  T1=(name)
   |
  PhD  T2=(name,degree)
   |
Employee T3=(name,degree,department)
```

Note that T2 redefines the attribute "name".

6.4 Deletion of a node

Deleting a node corresponds to dropping a class in the class DAG. We allow the deletion of a class in any position in the class DAG. Deleting a class with this assumption is an involved operation. The semantics of the update is the following:

Deletion of a class C corresponds to the deletion of all edges to the subclasses of C (if any), the deletion of all edges to the superclasses of C and the deletion of the class C from the DAG. Because of the possible different semantics of the delete-edge update we have different alternatives when deleting a class. The alternative semantics of deleting a class are obtained by composition of the different semantics resulting from deleting edges. The update will have parameters to specify such alternatives. We do not indicate them here, but instead define some constraints when deleting a class:

a) Deleting a class can only take place if its class extension is empty.

b) No system-defined classes can be deleted;

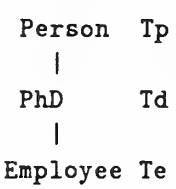
c) Deletion of a class can only take place if the class is not referred to in method signatures or in types of other classes.

d) A class cannot be deleted when its only superclass is the system class OBJECT and the class is not a leaf in the DAG.

Condition a) restricts the deletion of a class C to the case where there are no objects associated to the class. It also implies no automatic coercion to other class types in the schema (e.g. a superclass of C) of objects belonging to the extension of C. We assume the existence of explicit updates to delete objects or to change the object class types (cf. Section 9). Condition b) is obvious. Condition c) avoids type inconsistency for method signatures and attribute types. Condition d) preserves the DAG consistency.

Deletion of a class C may create behavioral inconsistency. In particular all other methods in the schema which depend on class C, i.e. either by calling methods of C or using directly the type T of C (if T is defined "public") must be marked unsafe. The dependency information introduced in Section 4.1 helps in detecting unsafe methods.

Example: Consider the schema below:



Deletion of the class PhD can be expressed in different ways, depending on the resulting schema we want to obtain (we consider the extension of the class PhD empty). We list as an example one of them:

```
  delete_class <PhD> =

T1: <(drop_edge <PhD-Employee>),(drop_edge<Person-PhD),
      (remove <PhD>)>.
/* the class Employee is made a subclass of OBJECT, its type is unchanged  */
```

The deletion of the class PhD can also be specifed so that the type of Employee is changed (i.e. it loses the attributes defined in PhD).

7 Related Work

Structural consistency is assured by all systems supporting a mechanism for changing the schema. To compare the different solutions is not always easy because there is no underlyng common model for such systems. The pioneering work in the field of schema manipulation in the context of object-oriented database systems is the one proposed for the Orion database system. In [Ba87a,Ba87b] a complete taxonomy of schema update operations is presented and for each update the semantics is given with a set of rules to preserve some "invariants" of the schema. Our definition of structural consistency implies the invariants of Orion[Ba87a,Ba87b]: 1) Class lattice invariant, 2) Distinct name invariant, 3) Distinct identity (origin) invariant, 4) Full inheritance invariant, 5) Domain compatibility invariant.

The O_2 and Orion data models differ in a number of characteristics; we briefly list them here. The existence in O_2 of *types* as well as classes is not present in Orion. The O_2 typing system can be viewed in the schema update context as a set of additional constraints which modify the Orion invariants 4. and 5. In fact in O_2 , methods and attributes in an inheritance hierarchy have to be type-compatible. This is taken into account in the definition of a consistent schema. Both systems allow multiple inheritance. O_2 has a set inclusion semantics for subtyping. Orion has no set inclusion semantics. In O_2 multiple inheritance conflicts are solved by the designer; when there is an ambiguity in the inheritance of a method (attribute),the designer must specify which one he/she wants to inherit or redefine it. As opposed to O_2, Orion automatically solves ambiguities. The system maintains an ordering among the superclasses which overrides ambiguites in the inheritance of methods. This difference modifies the Orion invariant 3. The O_2 distinction between objects and values is also present in Orion. In this system, however, the notion of a complex value is implemented as a dependent object (composite object). That is, non shared values are still objects

with a constraint enforcing their privacy. Because of these differences in the two data models the update semantics are also different. In particular class updates in O_2 provide parameters to allow the designer to specify the semantics he/she wants to have. Orion adopts a default-approach semantics for update operations; for example, when removing an edge $< S - C >$ in the DAG if the class C becomes disconnected, then it is automatically re-connected to all superclasses of S. The attributes of the type of class C which are inherited from S and not locally redefined in C are lost. No other possibilities are allowed.

GemStone [PeSt87] follows an approach similar to Orion to ensure structural consistency when updating the schema. Invariants are also used there; the major differences with the O_2 system are that Gemstone does not support types, has only objects and no values, and supports simple inheritance. Therefore structural problems are simpler with respect to systems which allow types such as O_2 and Vbase [AR87], and multiple inheritance, such as Orion and O_2. Vbase has types , but only supports single inheritance.

The issue of behavioral consistency is not considered in many object-oriented database systems. To our knowledge Orion [Ba87a,Ba87b,Kim88], GemStone[PeSt87], Vbase do not support any mechanism for dealing with behavioral inconsistency in the database. In particular, in Orion it is the user's responsibility to avoid (or cope with) the possible occurrence of dangling references when updating composite objects. In fact, in Orion it is possible to delete a class when the class extension is not empty. All objects of the deleted class are logically deleted. If the deleted object is part of another object (composite objects) then all dependent objects are recursively deleted.

The Encore [SkZd86,Zd87,SKZd87] data model is similar to that of O_2 as it provides types and multiple inheritance. Encore ensures structural consistency and provides an interesting mechanism for behavioral consistency based on versions of types and exception handlers. A change to a type creates a new version of the type. Exception handlers are associated to versions of types in order to allow different instances of different types to be used uniformly. Errors resulting from methods using a method or an attribute which are undefined or unknown are processed by the appropriate handlers.

Versions of schema are instead proposed in Orion [KiCh88]. This solution has the advantage of allowing different programs to use different versions of the schema.

In our current solution when updating an schema we did not consider creating versions, either at schema or at object level.

Other OODBM systems, such as Iris, Zeitgeist, and Trellis/Owl although not a true OODBS, do support schema changes [Pan88], but we are not aware of the type of updates they provide and the way they implement them.

A different approach is taken in the LOGRES project [CCCTZ90], an integration of the object-oriented data modelling paradigm and of the rule-based approach, where updates are expressed in modules by appropriate logical rules.

8 Conclusions and Future Work

We have outlined in this paper the definition of a set of primitives for schema manipulation for an object-oriented database system. In the paper we have defined a set of basic schema updates, and two kinds of consistency (structural and behavioral) which are desirable when performing a schema update.In particular, this proposal has influenced the schema update mechanism offered by the O_2 object-oriented database system.

More work is needed to improve our current approach. In particular it would be desirable to:

- Define a set of high-level restructuring operations on the schema. We can view the schema

tool manager as providing the basic mechanisms that can be used to build on top of it a more sophisticated object-oriented design tool, such as a tool which will enable compositions of classes, creation of superclasses given a set of classes, and so on. This design tool would only be used to modify the schema with no actual database (i.e. pure design phase). Interesting restructuring operations based on re-writing rules for schemas defined as a tree have been proposed in [AbHu88]. The same techniques could perhaps also be applied to schemas defined as a graph.

- Design an intelligent advisor tool for schema manipulation and in general a set of tools for helping the designer in performing schema updates. The advisor would participate in schema-update manipulations, including testing and validation, and also would suggest to the user a set of possible update alternatives (which may imply an interactive reply from the user) whenever some inconsistency may arise. At update time, if a test shows that an update is illegal, then the advisor augments the update with secondary updates to obtain a transaction that is legal. This process is recursive because the secondary updates might require tertiary updates, etc. The consistency advisor tool transforms an illegal update into an and/or tree of updates from which the user can construct a legal *transaction* [Zic89] that includes the original illegal update. The advisor produces a set of hints which the user does not have necessarily to follow. However, the advisor does not produce an exhaustive list of update alternatives. The criteria for generating alternative update strategies are in transforming an illegal update into a new one which does not produce undesired inconsistencies. The notion of transaction update has also been introduced in [GPZ88] in the context of the view update problem.

- Allowing type transformation with incomplete types and object values [Zic89], [GZ88].

Acknowledgements
The members of the Altair group provided useful comments on an earlier version of this paper.

9 References

[AbHu87] S. Abiteboul, R. Hull, "IFO; A Formal Semantic Database Model", *ACM TODS*, vol. 12, No. 4, December 1987.

[AbHu88] S. Abiteboul, R. Hull, "Restructuring Hierarchical Database Objects", *Theoretical Computer Science*, 62, 1988.

[AbKa89] S. Abiteboul, P. Kanellakis, "Object Identity as a Query Language Primitive", Proc. *ACM SIGMOD*, Oregon, June 1989.

[AKW89] S. Abiteboul, Kanellakis P., Waller E., "Method Schemas" (extended abstract), October 1989, (submitted for publication)

[AR87] T. Andrew, C. Harris,"Combining Language and Database Advances in an Object-Oriented Development Environment", Proc. *OOPSLA* Conference, October, 1987.

[Ba87a] .J. Banerjee et al,"Semantics and Implementation of Schema Evolution in Object-Oriented Databases", Proc. *ACM SIGMOD*, 1987.

[Ba87b.] J. Banerjee et al., "Data Model Issues for Object-Oriented Applications", *ACM TOOIS*, vol.5,No.1, January 1987.

[Ban88] F. Bancilhon, "Object Oriented Database Systems", Proc. 7th *ACM Symposium on the Principles of Database Systems*, Austin, Texas, March 1988.

[CCCTZ90] Cacace F., Ceri S., Crespi-Reghizzi S., Tanca L., Zicari R., "Integrating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm", proc. *ACM SIGMOD* Conference on Management of Data, Atlantic City, May 1990 (to appear).

[DelC89] C. DelCourt, "The Integrity Consistency Checker (ICC): Detailed Specifications", GIP Altair, October, 1989.

[DeZi89] C. Delcourt, R. Zicari, "Preserving Structural Consistency in an Object-Oriented Database", report GIP Altair, July 1989.

[Del89] C. Delobel,"A Formal Framework for the O2 Data Model", report GIP Altair, to appear.

[GPZ88] G. Gottlob, Paolini P., Zicari R., " Properties and Update Semantics of Consistent Views", *ACM TODS*, Vol. 13, No. 4, December 1988.

[GZ88] G. Gottlob, R. Zicari, "Closed World Databases Opened through Null Values", Proc. 14th *VLDB 88*, Los Angeles, August 1988.

[HK89] S Hudson, King R., "Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System", *ACM TODS*, Vol. 14, No.3, September 1989.

[HTY89] R. Hull, Tanaka K., Yoshikawa M., "Behavioral Analysis of Object- Oriented Databases: Method Structure, Execution Trees, and Reachability", (extended abstract), proc. *FODO89*, 1989.

[HVZ90] G. Harrus, Velez F. Zicari R., "Implementing Schema Updates in an Object-Oriented Database System:A Cost Analysis",Altair report (in preparation)

[KiCh88a] W. Kim, and Hong-Tai Chou,"Versions of Schema for Object-Oriented Databases" , Proc. 14th *VLDB 88*, August 1988, Los Angeles.

[Kim88] W. Kim et al., "Integrating an Object-Oriented Programming System with a Database System", Proc. *OOPSLA*, 1988.

[LeRi88] C. Lecluse, P. Richard, "Modeling Inheritance and Genericity in Object-Oriented Databases", Proc. *ICDT-88*, Bruges, August, 1988.

[LeRi89a] C. Lecluse, P. Richard, "The O_2 Database Programming Language", in Proc. 15th *VLDB 89*, Amsterdam, August 1989.

[LeRi89b] C. Lecluse, P. Richard, "Modeling Complex Structures in Object-Oriented Databases", Proc. *ACM PODS*, 1989.

[LRV88] C. Lecluse, P. Richard, F. Velez, "O_2, an Object-Oriented Data Model", Proc. *ACM SIGMOD*, Chicago, Illinois, June 1988.

[Mai89] D. Maier, private communication, October 1989.

[Pan88] Report on the Object-Oriented Database Workshop: Panel on Schema Evolution and Version Management, *SIGMOD RECORD*, Vol.18, No.3, September 1989.

[PeSt87] D.J. Penney, J. Stein, "Class Modification in the GemStone Object-Oriented DBMS", *ACM OOPSLA*, October 1987.

[SkZd86] A.H. Skarra,S.B. Zdonik, "The Management of Changing Types in an Object- Oriented Database", *ACM OOPSLA*, September 1986.

[SkZd87] A.H.Skarra,S.B.Zdonik, "Type Evolution in an Object-Oriented Database", in *Research Directions in Object Oriented Systems*, MIT press,1987.

[Vel89] F. Velez et al.,"The O_2 Object Manager: an Overview", in Proc. 15th *VLDB 89*, Amsterdam , August 1989.

[Zic89] R. Zicari, "A Framework for Schema Updates in an Object-Oriented Database System", *"The O_2 Book"* (F. Bancilhon, C. Delobel, P. Kanellakis, eds.), Morgan Kaufman to appear; also as GIP Altair 38-89 report, October 1989.

[Zd87] S.B. Zdonik, "Can Objects Change Types? Can Type Objects Change?", (extended abstract), *Workshop on Object-Oriented Databases*, Roscoff, September 1987.

The need for a DML: Why a library interface isn't enough

Jack Orenstein
Eugene Bonte

Object Design, Inc.
One New England Executive Park
Burlington, MA 01803

Introduction

An OO DBMS is an ideal platform for the development of complex applications that store and manipulate shared, persistent data. There are two alternatives to an OO DBMS, 1) a user-developed, ad hoc file system, and 2) a relational DBMS. The cost of developing and maintaining an ad hoc file system is becoming increasingly difficult to justify now that off-the-shelf OO DBMSs provide superior functionality and performance. Many application developers who use ad hoc file systems considered and rejected RDBMSs because of performance problems and because of the difficulty of working with the database through a host language interface.

An application developer can work with an OO DBMS through a variety of interfaces. In this paper we consider programmatic interfaces only. All currently available OO DBMSs offer library interfaces. There are clear benefits to a library interface. It is portable across languages and requires no special compiler support, e.g. a pre-processor. Application development tools, (e.g. for browsing, schema management, database reorganization, report generation), are well-supported by such an interface.

However, there is one aspect of database management that is not amenable to a library interface. This has to do with associative access. The great success of relational DBMSs (for "traditional" applications) is due, in large part, to the facilities for associative access. These facilities include:

- A non-procedural language for specifying queries, (SQL, for relational DBMSs).
- The ability to dynamically add and remove indexes which speed up certain queries.
- Maintenance of indexes, in response to updates to the database.

- Query optimization - the automatic selection of an execution plan for each query.

This approach permits *correctness* of the application to be separated from performance concerns. Correctness has to do with the formulation of SQL statements. Performance is controlled by adding indexes that speed up certain kinds of retrieval. The query optimizer selects from the available indexes and generates an execution plan that coordinates the use of index lookups and index scans to compute the answer to the query. The SQL query does not have to be modified as indexes are added or dropped in response to performance requirements.

We believe that associative retrieval is not well-supported by a library interface. This capability is best supported by a data manipulation language (DML), with an embedding defined for each host language. In the rest of this paper, we show why associative retrieval is difficult to support through a library interface, and demonstrate the advantages of an embedded DML approach. This leads us to conclude that embedded DML standards are required in addition to library interface standards. We also discuss the relationship of an OO DML to SQL.

Data Manipulation

Some current OO DBMS library interfaces offer set or collection classes. These are very minimal implementations, as they do not support associative retrieval with query optimization. A rudimentary form of associative access is currently available through hash table and b-tree object classes (to be referred to generically as indexes). The interface to these object classes might be found as part of an OO DBMS library interface. These object classes support iteration, update, and rapid lookup of (key, value) pairs given a key. Simple queries can be handled easily using these object classes.

There is no connection between sets and indexes, so the use of indexes to optimize retrievals from sets is in the hands of the programmer. This is especially unpleasant for complex queries which may involve the use of multiple indexes. In these cases, the results of one index lookup feed into lookups on other indexes. It may be necessary to record intermediate results and to performs unions and intersections on these intermediates, corresponding to conjunctions and disjunctions in queries. For example, consider an MCAD application dealing with parts. Each part has a name, a

weight, and a set of subparts, and there is a set of all parts. Consider the following query:

```
Find parts whose weight is less than 100 that
contain a "frammis".
```

There are two indexes that could be of use here, an index permitting rapid lookup of parts given weight, and an index from part name to containing part. (E.g. if a widget contains a frammis, then the second index maps from the name "widget" to the part whose name is "frammis".) To process the query, each index would be searched, yielding two sets of part ids. The ids common to both sets would then be returned as the answer to the query.

It is clearly preferable to be able to specify this query at a higher level, without having to deal with indexes explicitly. This simplifies the formulation of queries, and, as in an RDBMS, insulates the query from changes in indexing decisions. This raises the question of how to specify queries through a library interface. There could be a function in the interface that takes a string containing a DML form of the query and returns the result, e.g.

```
query("parts[ weight < 100 and
        subparts[ name = 'frammis'] ]")
```

This approach has a number of problems. First, in practice, the constants 100 and "frammis" are likely to be stored in program variables, and it is a nuisance to have to create a string that incorporates the variables' values. If the query involves a more complex selection, e.g., involving function invocations, then this approach breaks down, limiting the power of the DML. Second, the type of the query result depends on the query, so this approach is problematical for strongly-typed languages, which require type information at compile-time. Third, this approach is not conducive to compile-time analysis of the query.

Another approach to a library interface DML is to offer a collection of functions for constructing a "parse tree" representing the query, e.g.

```
query(parts,
      and(lt(weight, 100),
          query(subparts, eq(name, "frammis"))))
```

This expression would yield an object that can then be passed to an evaluation function. This approach has the advantage of separating optimization from evaluation, but otherwise has many of the drawbacks of the previous approach.

The simplest interface, from the programmers point of view, is a DML embedded in the host language. In an earlier paper [ATWO90] we described this approach in more detail and gave examples of a C++ DML. There are strong resemblances to other C++ DMLs [AGRA89, BLAK90] and to a Smalltalk-based DML [MAIE86]. The similarities among these four independently developed DMLs points to the viability of this approach. Example (C++):

```
set<part> s;
int w = 100;
string n = "frammis";
s = parts[weight < w and
          subparts[ name = "frammis"]];
```

This approach offers the greatest flexibility in integrating DML and the host language. Queries may be used in any context, and may themselves contain embedded expressions of the host language. DML capabilities overlap host language capabilities (e.g., both languages have variables and logical expressions), so host language syntax can be used to provide much of the DML syntax. This simplifies use of the DML and further contributes to close integration. There is no need to establish a binding between DML variables and host language variables; again, the host language variables can be used in the DML.

One of the most serious drawbacks of host language interfaces to RDBMSs is the "impedance mismatch" - queries to the database return sets, but the host language is not equipped to deal with sets except by iteration. In OO host languages, such as C++ and Smalltalk, this problem disappears thanks to set or collection object classes, which are already present in OO DBMS products.

Indexes

As discussed above, OO DBMS library interfaces provide set and index classes for associative retrieval. This is not a satisfactory interface because the programmer has to perform all index manipulations explicitly. If a DML is to be supported, through a library interface or a language binding, then the indexes are still present, but are much less obtrusive. The application programmer can ask for indexes to be added to and removed from sets, and can then forget about them. Index maintenance - modifying indexes in

response to database updates, and the use of indexes in the evaluation of queries are all hidden, carried out by the DBMS.

Adding and dropping indexes are simple to do, with either a library interface or through a language binding. Index maintenance, however, is much more complicated. It is not a simple matter to identify actions that require index updates, and then figure out what indexes need to be updated. In the previous example, a set of parts had two indexes, 1) from weight to part id, and 2) from part name to id of a containing part. If a part's weight is updated, then the index on part weight needs to be modified. Suppose that part p has weight w, and the weight is changed to w'. The index on part weight has a (w, p) entry, keyed by w. This needs to be replaced by (w', p). It is not sufficient to simply update w in place. Since the index is keyed by w, the (w, p) entry has to be removed, and the (w', p) has to be inserted.

The other index is more complicated because it goes through a subpart relationship. Suppose that part p has a subpart s named n. Then the index has an (n, p) entry. Changing a part's name must trigger an index update. If the name of s is changed to n', then (n, p) must be replaced by (n', p), (again, by removal and insertion, not update in place). An index update must also occur if s is removed as a subpart of p. In this case, the correct update is to remove (n, p), and not insert anything.

These "database updates" are simply assignment statements in an OO DBMS host language, e.g.

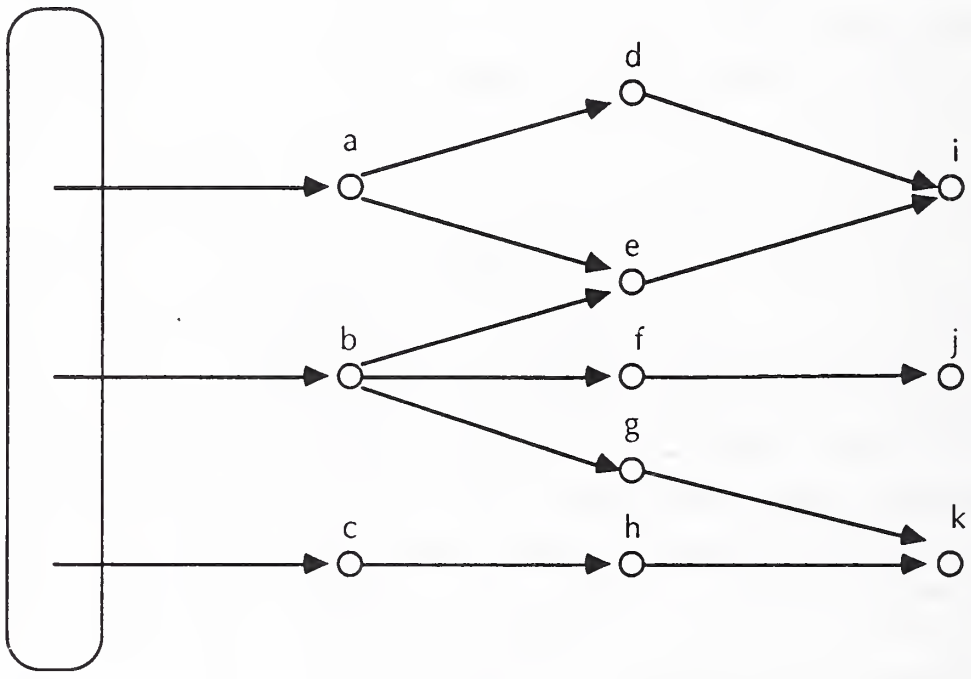
```
part* p;
...
p->weight = 120;
```

The index update needs to be triggered somehow. For a library interface, a function call is required, e.g.

```
index_update(parts, p->weight, 120);
p->weight = 120;
```

The index_update function has arguments indicating the set which has the index, the index key (p->weight), and the new value of the key. Following the index update, the actual assignment is done. Actually, this is a simplification, since there needs to be some way to identify which index of parts needs to be updated, because there may be other sets with indexes on part weight, and because there may be sets with indexes on part weight which do *not* need to be updated, (e.g. if the set does not contain p). The

rules describing what updates are needed are complex, and this is an extremely simple case! In general, index maintenance requires a graph reachability computation. Example:



This diagram shows a set containing objects {a, b, c}. These objects contain sets of other objects, a contains {d, e}, b contains {e, f, g}, and c contains {h}. Each of the objects d - h are connected to one of object i, j, and k. Suppose that the set has an index on these paths. The index contains

key	value
i	a
i	b
j	b
k	b
k	c

Updates, which seem similar to one another, can have different consequences for the reachability of other objects, and therefore, different index update actions. For example, if the b-f link is broken, then j becomes unreachable, so the (j, b) entry should be removed. But if the a-e link is

broken, i is still reachable (via d), so the index does not have to be modified. Keeping track of these dependencies, and computing reachability is not the sort of thing that users should have to do - the system can do this efficiently and correctly [MAIE86].

This is not a satisfactory situation. The library interface to index maintenance is too difficult to use. This is compounded by the fact that errors in index maintenance are subtle and may easily go undetected. Finally, maintenance is a nightmare. As indexes are added and removed, all the index maintenance code has to modified.

The index maintenance interface could be automated somewhat, but only if the schema is modified to accomodate fields within objects to keep track of indexes that need to be updated. The complexity of the interface does not go away completely, as these fields would have to be explicitly declared by the user.

There is a clear need for some support to deal with the index maintenance problem. This support can come from the DML pre-processor. For each updates to an object property that serves as index key, index maintenance code is generated. It is safer and simpler to rely on the pre-processor to generate the required code than to do it by hand. With such support, the user can write an update such as

```
p->weight = 120;
```

and all required index maintenance will take place.

Query Optimization

Three different approaches to DML have been proposed:

1. Pass DML as a string to a query function.
2. Construct a tree representing a query and pass it to an evaluation function.
3. Embedded DML.

#1 and #2 are library-based approaches. Query optimization is possible with any of these approaches, but only #3 permits compile-time query analysis.

Compile-time query analysis has significant advantages. It provides immediate feedback about the correctness of the query - syntax errors can be detected at compile-time. (Type safety is also guaranteed in languages where that is important, e.g. C++). Of course, additional information for optimization is available at run-time, but performance is improved by moving as much work to compile-time as possible. For example, at compile-time, multiple strategies could be generated and compiled. At run-time, examination of the database (e.g. for presence of indexes) indicates which strategy should be selected. Most of the expensive state-space search was carried out at compile-time, leaving a much smaller space to explore at run-time.

What about SQL?

A DML integrated with the host language has many advantages over a library interface DML:

- The DML itself is simpler, because it can take advantage of host language constructs (e.g. variables, logical expressions). The DML is also more expressive since any host language expression can appear inside a query, e.g. in a selection predicate.
- Compile-time query analysis is possible. This leads to type safety for the DML. Also, the optimizer can do much of its work at compile time instead of at runtime.
- Index maintenance can be hidden completely. Index maintenance through a library interface is extremely difficult to do correctly.

This integration can be accomplished by a DML pre-processor.

So far we have focussed on the issue of DML interfaces. We now discuss the form of the DML. The DML should be standardized, so it is natural to question whether another standard query language is necessary, given the dominance of SQL in the RDBMS world. Could SQL be used for OO DBMSs too?

Some existing OO DMLs bear superficial resemblances to SQL (especially [BLAK90]), but there are major semantic differences. A relational database consists of a set of relations storing tuples of values. A typical query

joins relations based on matching values in different relations, selecting tuples that satisfy a predicate, and then discarding unneeded fields. Example:

```
PERSON(id#, name, age)
JOB(person, salary, title)

/* Find the names of people whose age is
   less than 30 and whose salary is greater
   than 100000 */

SELECT name
FROM PERSON, JOB
WHERE person = id# AND age < 30 AND salary > 100000
```

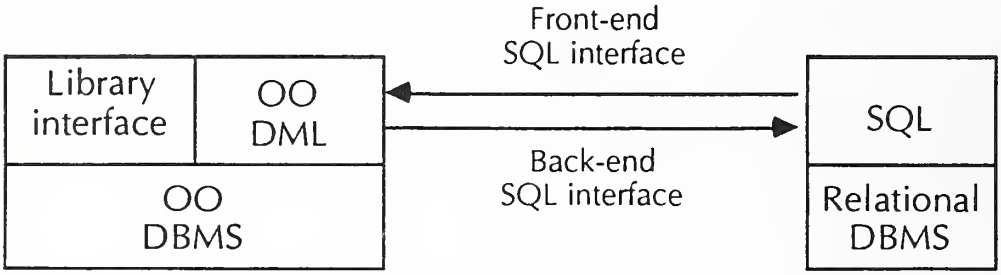
The connection between PERSON tuples and JOB tuples is based on matching values in the id# field of PERSON and the person field of JOB. Tuple values are permitted to be simple types only, e.g. numbers and strings.

By comparison, an OO DBMS schema contains fewer "top-level" constructs, and the objects stored may be structured. For example, an OO version of the above relational schema would typically contain a single set of person objects, and each person would store the id of a job object - a pointer. The explicit pointer can be thought of as a "pre-computed" relational join. I.e., the result of the join is stored, so the join (indicated by the person = id# term in the query) does not have to be spelled out.

Set-valued properties of objects are modeled very differently in the two models. In an OO DBMS, an object may have a set-valued property or attribute, e.g. part objects have subpart attributes. The type of subparts is set of parts. In a relational DBMS, these sets can't be represented directly, as set-valued tuple values. Instead, a new relation has to be introduced to store all the part-subpart relationships. The relation would have one tuple for each such relationship. I.e., the cardinality of the relation is the sum of the cardinalities of all the subparts sets. Viewed another way, the "embedded" subpart sets would, in a relational schema, be combined into a single (top-level, non-embedded) relation.

Because relational and OO DBMSs differ in the way they model applications, the DMLs should be different. For this reason, the primary DML of an OO DBMS should *not* be SQL.

However, it will be important for OO DBMSs to co-exist with RDBMSs, so SQL support is important. There are two forms of support that will be required, front-end and back-end.



"Front-end" SQL support is the ability to submit SQL queries toß an OO DBMS. This is important in order to make use of existing software and expertise to access an OO DBMS. Such a situation might occur in an engineering company where the design data has been moved into an OO DBMS. Meanwhile, the accounting and management software, which used to access an SQL database needs to access the same information, but now in an OO DBMS. "Back-end" support for SQL is the ability to use an OO DBMS, including the OO DML, to access data in an SQL database.

In both cases, there is a *view mapping* problem, comprising schema translation and DML translation. Many of the ideas that were developed to deal with heterogeneous relational and pre-relational databases can be applied. Since DML statements assume a specific schema, schema translation must take place before DML translation.

Summary and conclusions

One of the major reasons for the success of relational DBMSs is that they provide a high-level, non-procedural query language and the facilities to support it: indexes, automatic index maintenance, and query optimization. These features are not present in OO DBMS library interfaces. Rectifying this situation will be difficult because it is difficult to describe queries (except for some simple, special cases) through a library interface. The resulting interface has some of the drawbacks of programmatic SQL interfaces - they are ugly and difficult to use.

A library interface that supports index maintenance is easy to define, but extraordinarily difficult to use correctly. The problem is in knowing when index maintenance is required, and in knowing what maintenance actions are appropriate.

A better solution is to define DML capabilities which can be embedded in host languages in a way most appropriate to that host. The constructs of the host language are used where possible. Additional syntax may be required. Programs that use the DML would then go through a pre-processor, expanding the DML into procedural code, and inserting index maintenance actions where appropriate. This approach retains the best of both worlds: the generality of the host language and the convenience of a query language. We therefore recommend the development of standards for the DML semantics, and for each host-language embedding.

References

AGRA89 R. Agrawal, N. Gehani.
ODE (object database and environment): the language and the data model.
Proc. ACM SIGMOD, Portland, Oregon, (June, 1989).

ATWO90 T. Atwood, J. Orenstein.
Notes toward a standard object-oriented DDL and DML.
Preliminary Proceedings of the First OODB Standardization Workshop, Atlantic City, New Jersey (May, 1990).

BLAK90 J. Blakely, C. Thompson, A. Alashqur.
Strawman reference model for object query language.
Preliminary Proceedings of the First OODB Standardization Workshop, Atlantic City, New Jersey (May, 1990).

MAIE86 D. Maier, J. Stein.
Indexing in an object-oriented DBMS.
Proc. Int'l Workshop on Object-Oriented Database Systems, Pacific Grove, California, (Sept, 1986).

Foundations for Object-Oriented Query Processing

Karen C. Davis
Department of Mathematical Sciences
The University of Akron
Akron, OH 44325-4002
kcd@zippysun.math.uakron.edu

Lois M.L. Delcambre
The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504-4330
lmd@cacs.usl.edu

1 Introduction

The most significant obstacle to object-oriented query processing research is the lack of agreement on a data model, and thus on a formal foundation for query language, query optimization, and view definition and processing research. To overcome the data model problem, we adopt a high-level, conceptual model of complex objects along with a formally-defined, algebraic query language. The core of the query language is the ability to form subtypes/subclasses of existing types/classes by restricting inherited properties. A strength of this approach is that all attributes and values of a query result are derived from existing database objects (both schema and data); retaining connections to the schema provides logical access paths to data. The formalism supports provably correct logical query optimization in two forms: algebraic transformations and classification-related optimizations.

The object-oriented, algebraic query language introduced here provides relational algebra functionality in an object-oriented setting. The algebra provides operators which preserve object identity and allow complex objects to be manipulated as a whole, as well as operators which allow objects to be manipulated based on the values of their attributes. The ability to specify unstructured joins is provided via the cross product operator. The algebra also supports the ability to create new objects representing only the values derived from existing objects. Just as relational algebra is closed over relations, the algebra is closed over classes.

The denotational semantic specification of the algebra precisely defines the operators; it makes the types of the data model and the behavior of the operators over them explicit. The formal definition provides the basis for implementation as well as the basis for provably correct logical query optimization. The definition of the algebra contributes a formal description of both intensional and extensional query answers. Each query describes a class, which has intension (e.g., inherited properties and a membership definition) and extension (data values).

Any data model which supports complex objects can use the algebra presented here for query specification. It is especially well-suited for data models which support the isa relationship, properties between classes, and inheritance of properties. It is not necessary that the data model support definitional membership in classes (i.e., that the data model intension includes property restriction and set-theoretic specification of membership), since all classes in a data model of this nature can be treated as base classes.

The research sets the stage for integration of query optimization into a full-featured object-oriented database. The conceptual model utilized in this research for developing a high-level query algebra is not a full-featured object-oriented data model since it does not have facilities for specifying user-defined behavioral abstraction. However, since the algebra can be used to express information

such as access paths, the structural abstraction provided by the conceptual model is sufficient for exploring query language issues in object-oriented data models. The algebra proposed here can bridge the gap between implementations and query interfaces, since it provides object-oriented and value-oriented query power, support for views, and the basis for logical query optimization; this research represents a step toward developing theoretically-founded query languages for object-oriented database management systems.

To summarize our point of view, a minimal framework for a query language standard should address the following issues:

1. formal definition of a high-level data model, including the formal definition of a database and the syntax and semantics of associated query languages,
2. proofs of correctness of transformations, so that syntactic rewriting of query expressions are meaning-preserving according to the formal definition, and
3. functionality of relational query languages, enabling both value-based and object-based queries as well as closure over class types.

2 A Conceptual Model for Object-Oriented Databases

A minimal set of criteria for an object-oriented database system has been identified by Zdonik and Maier [ZM90] as follows:

1. database functionality: storage for persistent data, efficient access to data, concurrency control, recovery, etc.,
2. object identity: a unique identifier regardless of values of objects,
3. encapsulation of structure and behavior: access to objects is only allowed via an interface defined for the object, and
4. complex state: the ability to reference other objects.

Providing database functionality and enforcing encapsulation (items 1 and 3 above) are essentially database system features (details of implementation), although a conceptual data model should in no way preclude them. Support for object identity and complex state (items 2 and 4) must be addressed at the conceptual level.

Query processing, from a query writer's point of view, is concerned with the conceptual level. Physical details, such as whether data is stored or derived, are invisible to the user. Encapsulation can be preserved, since a user need only retrieve data visible through the interface defined for a class. Queries use methods that do not change the states of objects (*inspectors*, not *mutators* [ZM90]) to view the values of objects. In other words, the query language for an OODB can be a high-level query language based on the abstractions provided by a conceptual data model.

Our proposed conceptual data model, essentially a semantic data model (e.g. [HM81]), provides the following:

1. object identity,

2. complex state,

3. the isa relationship (set inclusion semantics), and

4. inheritance of properties (i.e., inspector methods).

The semantics associated with these features represent the rich intension associated with complex objects and they influenced the development of the query language presented here. The notion of a class hierarchy encompasses both the type and collection hierarchies of Zdonik and Maier [ZM90]. Since queries process collections, and all collections have types, they are related in the conceptual data model through a generalization/specialization hierarchy (the isa relationship), with set inclusion semantics.

The conceptual model provides automatic membership [CADF90] in classes. *Automatic* (or *definitional*) membership refers to membership in a class that is described by an intensional rather than extensional specification (e.g., enumeration or user-controllable membership). Automatic membership insulates the user from details, especially physical storage details.

For the purposes of the query language, definitional membership is supported via the modification of inherited properties. A property has a range class which serves as the domain for the values of objects under the property. A numeric constraint on the cardinality of the image set (how many values an object may have for a particular property) is used to express single-valued and multi-valued properties. Subclasses can be defined by restricting both the range class and/or the cardinality constraints of properties.

3 A Formal Query Language

The database state is composed of intensional information and extensional information. The intensional portion of the database state contains structural information about classes such as details of their properties and membership definitions. The formal definitions of the query operators include semantics for determining the inherited properties and membership definitions of query answers. The extensional portion of the database state includes the extension of classes (sets of object identifiers) and mappings of objects to their values (also called *images*) under properties.

Denotational semantics provides an elegant means for expressing the essential features of language semantics: the syntax of the language, the domains (environment) where the syntactic constructs have meaning, and the mapping from expressions in the language to meaningful results in the environment. For the query language, the environment is a database state, both intensional and extensional. Each each query operator creates a class with intension and extension *derived* from existing classes and query specifications. A query result can be viewed as a system-generated class, the details of which are contained in a modified database state.

The operators of the algebra are listed in Figures 1 and 2, along with an informal summary of their formal meaning. A query expression is denoted by e. The intensional effects of the operators (Figure 1) are described in terms of the inherited or derived properties of the query result and its

Operator	Properties	Membership Definition
$e_1 \cup e_2$	most general version of common properties	"or" membership definitions
$e_1 \cap e_2$	all properties with most specific common properties	"and" membership definitions
$e_1 - e_2$	properties of e_1	"difference" of membership definitions
ρ_{re} e	properties of e	"and" membership definition and restrict expression (re)
σ_{se} e	properties of e	"and" membership definition and select expression (se)
π_{pnl} e	properties in property list (pnl)	generated membership definition
$e_1 \times e_2$	two generated properties with e_1 and e_2 as range classes	generated membership definition

Figure 1: Intensional Query Results

membership definition. The extensional effects (Figure 2) describe which objects are in the query answer.

The first group of operators ($\cup$, $\cap$, $-$, ρ, σ) are the identity-preserving, subclass-forming operators. (The union operator actually produces a superclass of its argument classes, but the query result will be a subclass of the arguments' common superclass.) The restrict operator, ρ, is based on universal quantification; objects in a class defined by ρ will have entire images within the cardinality and range restrictions of the properties specified in the restrict expression. The select operator, σ, allows objects to be selected based on part of their images for properties. The remaining operators, project (π) and cross product ($\times$), form classes that are derived from existing classes; the members of these classes are new objects whose images are existing objects.

Operator	Extensional Effect
$e_1 \cup e_2$	union of object identifiers
$e_1 \cap e_2$	intersection of object identifiers
$e_1 - e_2$	difference of object identifiers
ρ_{re} e	subset of object identifiers based on entire image of objects for restricted properties
σ_{se} e	subset of object identifiers based on partial image of objects for selected properties
π_{pnl} e	new object identifiers assigned to distinct values
$e_1 \times e_2$	new object identifiers assigned to each pair of object identifiers in cross product

Figure 2: Extensional Query Results

4 Logical Query Optimization

The formal definition of the algebraic query language and database state allows for reasoning about what queries in the language *denote* (i.e., the meaning of the queries), so that syntactic transformations for logical query optimization can be proven to be meaning-preserving. If query results are identical (same intension and extension), then the equality holds. Denotations of expressions can be used to show that their intension and extension are the same. For defined classes, identical intension implies that extension is the same, since a non-trivial membership definition completely describes the objects that have membership in the class.

A classifier is a tool for computing subsumption relationships between classes; given a class and a class taxonomy, it determines the proper position for the class in the taxonomy [Lipk82, BFL83, SI83, FS84, Davis87, BBMR89, BGN89, DD89a, DD89b]. The formal framework for a classifier is provided by rules of inference for structural properties of the model [Davis87, DD89a, DD89b]. A classifier can be used to find logical access paths by pretending to classify classes that are described by queries [BGN89]. Through classification, the most specific subclasses satisfying the constraints of the query are used to construct the query results. Our research explores the simplification of query expressions within a query by identifying subsumption relationships, disjoint relationships, and redundant and inconsistent classes using a classifier [Davis90]. Examples are given in Figure 3.

The algebraic transformations developed and proven correct using the formal definition of the query language [Davis90] can be incorporated into a framework of guidelines for rewriting queries into more efficient forms; however, criteria for determining efficiency have yet to be developed. One possible optimization heuristic is that classifiable operators should be applied as soon as possible, similar to the concept of select migration in relational query optimization. Detailed investigation is a task for future study.

5 Analysis

OODBSs such as GemStone [MOP85, MSOP86], Iris [Fish87, Fish88], O_2 [BBBD88, LRV88], and ORION [BCGK87, BKK88] present no obstacles to using this conceptual model for query processing. Each supports a class hierarchy that allows structure and behavior to be inherited.

Query	Inferred by the Classifier	Simplified Query
$e_1 \cap e_2 \cap e_3$	$isa(e_2, e_1)$	$e_2 \cap e_3$
$e_1 \cap e_2$	$disjoint(e_1, e_2)$	$\emptyset$
$e_1 \cup e_2$	$redundant(e_1, e_2)$	e_1 (or e_2)
$e_1 \cap e_2$	$inconsistent(e_1)$	$\emptyset$
$e_1 \cup e_2$	$inconsistent(e_1)$	e_2

Figure 3: Query Simplification Using the Classifier

None of them support automatic membership, but since queries are definitional in nature, query processing against these OODBSs are supported because queries induce an isa relationship with inheritance and definitional membership.

With regard to query languages, although there are a number of object-oriented database management systems currently under development, there is no consensus on a single data model; implementation work has generally preceded any formal treatment of the model [Banc88]. Research on object-oriented query languages is just emerging. Systems such as GemStone [MOP85, MSOP86] and O_2 [BBBD88, LRV88] have been implemented without a query processing component. ORION has a query language with some cumbersome features, for example, the project operator which projects only one or all properties [BCGK87, BKK88]. Iris has a query language, OSQL, which relies on relational algebra as a computational model [DKL85, LK86, Fish87, Fish88]. The operators of the Encore Query Algebra [SZ89] and the Object Algebra [Osbo88] are similar to those of the algebra presented here, and in some cases have richer syntax and semantics (e.g., a more flexible select operator). In the Encore Query Algebra, all query results are tuple types, and in the Object Algebra, queries may result in sets, aggregates, or strongly-typed sets. In contrast, the operators presented here produce abstract classes as a result, so the algebra is closed over abstract classes. That is, queries can involve arbitrarily nested operators, with each intermediate result yielding an abstract class.

The subclass-forming operators ($\cup, \cap, -, \rho, \sigma$) described here allow complex objects to remain intact and retain their identity. These operators create new subclasses whose membership is derived from that of existing objects in abstract classes. This is in contrast to Encore's Query Algebra [SZ89], where new object identifiers are always assigned to objects, even for set-theoretic queries where the original identifiers could be used. The disadvantage of generating new object identifiers for every query result is that answers to queries are disconnected from the classes in the schema. In our approach, queries and subexpressions of queries which produce true subclasses are retained at the appropriate location in the isa hierarchy of the database schema. This approach retains the strengths of the schema, one of which is exploiting the isa relationship for providing logical access paths to data.

References

[BBBD88] Bancilhon, F., G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lecluse, P. Pfeffer, P. Richard, and F. Velez, "The Design and Implementation of O_2, an Object-Oriented Database System," in [Ditt88].

[BBMR89] Borgida, A., R.J. Brachman, D.L. McGuinness, and L.A. Resnick, "CLASSIC: A Structural Data Model for Objects," *Proceedings of the 1989 SIGMOD Conference*, Portland, Oregon, 1989.

[BCGK87] Banerjee, J., H.T. Chou, J. Garza, W. Kim, D. Woelk, N. Ballou, and H.J. Kim, "Data Model Issues for Object-Oriented Applications," *ACM Transactions on Office Information Systems,* January, 1987.

[BFL83] Brachman, R.J., R.E. Fikes, and H.J. Levesque, "KRYPTON: A Functional Approach to Knowledge Representation," FLAIR Technical Report No. 16, Fairchild Laboratory for Artificial Intelligence Research, Palo Alto, CA, May 1983.

[BGN89] Beck, H.W., S.K. Gala, and S.B. Navathe, "Classification as a Query Processing Technique in the CANDIDE Semantic Data Model," *Proceedings of the Fifth International Conference on Data Engineering*, Los Angeles, February, 1989.

[BKK88] Banerjee, J., W. Kim, K.-C. Kim, "Queries in Object-Oriented Databases," *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, February, 1988.

[CADF90] The Committee for Advanced DBMS Function, "Third-Generation Database System Manifesto," Memorandum No. UCB/ERL M90/28, Electronics Research Laboratory, University of California, Berkeley, CA, April, 1990.

[Davis87] Davis, K.C., "The Theoretical Foundation for Inferencing on a Semantic Schema," M.S. Thesis, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA, June 1987.

[Davis90] Davis, K.C., "A Formal Foundation for Object-Oriented, Algebraic Query Processing," Ph.D. Dissertation, The Center for Advanced Computer Studies, University of Southwestern Louisiana, Lafayette, LA, May 1990.

[Ditt88] Dittrich, K., ed., *Advances in Object-Oriented Database Systems: Second International Workshop on Object-Oriented Database Systems,* Bad Munster am Stein, West Germany. Also appears as *Lecture Notes in Computer Science,* No. 334, Springer Verlag, 1988.

[DKL85] Derrett, N., W. Kent, and P. Lyngbaek, "Some Aspects of Operations in an Object-Oriented Database," *Database Engineering*, Vol. 8, No. 4, December, 1985.

[DD86] Dittrich, K. and U. Dayal, editors, *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September 23-26, 1986.

[DD89a] Delcambre, L.M.L, and Davis, K.C., "Automatic Validation of Object-Oriented Database Structures," *Proceedings of the Fifth International Conference on Data Engineering*, Los Angeles, February, 1989.

[DD89b] Delcambre, L.M.L, and Davis, K.C., "The Design and Validation of Object-Oriented Schemas," CACS Technical Report No. TR-89-6-2, submitted for publication.

[Fish87] Fishman, D., *et al.,* "Iris: an Object-Oriented Database Management System," *ACM Transactions on Office Information Systems,* Vol. 5, No. 1, January, 1987.

[Fish88] Fishman, D., *et al.,* "Overview of the Iris DBMS," Database Technology Department, Hewlett-Packard Laboratories, Palo Alto, CA, 94304, June, 1988.

[FS84] Finin, T., and D. Silverman, "Interactive Classification as a Knowledge Aquisition Tool," *First International Workshop on Expert Database Systems,* 1984.

[HM81] Hammer, M., and D. McLeod, "Database Description with SDM: A Semantic Database Model," *ACM Transactions on Database Systems,* Vol. 6, No. 3, Sept. 1981.

[Lipk82] Lipkis, T., "A KL-ONE Classifier," *Proceedings of the 1981 KL-ONE Workshop,* edited by Schmolze, J.G., and R.J. Brachman, BBN Report No. 4842, Bolt Beranek and Newman Inc., 1982.

[LK86] Lyngbaek, P., and W. Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems," in [DD86].

[LRV88] Lecluse, C., P. Richard, F. Velez, "O_2, An Object-Oriented Data Model," *Proceedings of the ACM SIGMOD 1988 International Conference on Management of Data*, Chicago, IL, 1988.

[MOP85] Maier, D., A. Otis, and A. Purdy, "Object-Oriented Database Development at Servio Logic," *Database Engineering*, Vol. 8, No. 4, December, 1985.

[MSOP86] Maier, D., J. Stein, A. Otis, and A. Purdy, "Development of an Object-Oriented DBMS," *Proceedings of the First ACM OOPSLA Conference,* Portland, OR, Sept., 1986.

[Osbo88] Osborn, S.L, "Identity, Equality, and Query Optimization," in [Ditt88].

[SI83] Schmolze, J.G., and D. Israel, "KL-ONE: Semantics and Classification," *Research in Knowledge Representation for Natural Language Understanding, Annual Report September 1982 - August 1983,* BBN Report No. 5421, Bolt Beranek and Newman Inc., 1983.

[SZ89] Shaw, G., and S. Zdonik, "An Object-Oriented Query Algebra," *Database Engineering*, Vol. 12, No. 3, 1989.

[ZM90] Zdonik, S.B., and D. Maier, "Introduction," in *Readings in Object-Oriented Database Systems,* Morgan Kaufmann, 1990.

Inheritance and Generalization in Intelligent SQL

by
Setrag Khoshafian
Roger Blumer
Razmik Abnous

Ashton-Tate Corporation
2033 North Main Street
Walnut Creek, CA

As computers become more and more powerful and interconnected, the computer software industry is being pulled more and more towards integration of products, across both different platforms and different application domains. Users often need to access data on several different machines, each with their own operating system. Applications like word processors are linked to spreadsheets which are linked to databases.

The connection between these different platforms and different applications is shared data. This data must be persistent and must be concurrently shared by these applications. Persistency and concurrency are among the basic functions of database management systems (DBMSs); thus, DBMSs are an extremely important part of product integration. Users need DBMSs that can both easily model complex real-world relationships AND easily share data across different systems. The relational database of the 1980's do not provide these features. Intelligent databases achieve both of these objectives.

Intelligent databases generally include an object-oriented database model that allows direct representation of real-world models, support for declarative rules to express semantic relationships among objects, and strong support for text, image, voice, animation, and video. Intelligent SQL is part of a DBMS that has all of these attributes, and also includes ways to access databases from other machines and from other vendors. Several aspects of the Intelligent SQL language have been previously presented to the Object Oriented Database Task Group (OODBTG) in May 1990. This paper will present a detailed description of the object-oriented concept of *inheritance* and *generalization*. In particular, it will show how these concepts can be used to construct global distributed database schema in Intelligent SQL.

Inheritance

Inheritance allows a natural model of the real world to be represented in the database. It allows new objects to be built on top of existing objects. People are used to abstracting and classifying information in inheritance hierarchies. For example, people think of mammals as a subclass of vertebrates, dogs as a subclass of mammals, and terriers as a subclass of dogs (Cox, 1984). Inheritance allows developers to easily extend an application by specializing existing pieces of their application. This is known as top-down development.

Inheritance in Intelligent SQL is composed of two complementary aspects: specialization and generalization. Specialization can be used to form table hierarchies in a top-down fashion. Subtables can be defined in terms of existing tables. Generalization, on the other hand, is a bottom-up approach. Supertables are constructed from the common attributes of several existing tables.

Specialization

Intelligent SQL allows specialization in two ways: through adding new columns to the supertable (horizontal modification) and through restricting existing columns of the supertable (vertical modification). A subtable can also inherit columns from several supertables (multiple inheritance). As an example of horizontal modification, consider the tables Persons, Staff, and Students:

```
CREATE TABLE Persons (
            Name        CHAR(20),
            Age         INTEGER,
            City        CHAR(20),
            State       CHAR(2))

CREATE TABLE Staff
      SPECIALIZES Persons    (
            Salary      FLOAT,
            OfficeNo    CHAR(30))

CREATE TABLE Students
      SPECIALIZES Persons    (
            Major       CHAR(20),
            Advisor     CHAR(20),
            GPA         FLOAT)
```

re Staff inherits Name, Age, City, and State from Persons, and adds two new umns Salary and OfficeNo. The table Students inherits all of the columns in rsons and adds the column extensions Major, Advisor, and GPA (Grade Point erage). These are examples of single inheritance in its simplest form (see Figure). The full syntax for specialization of tables is:

```
CREATE TABLE <table name>
        SPECIALIZES <restricted table name list>
        [CORRESPOND <table attributes list>]
        [AS <column name list>]
        [(<table elements>)]
```

ere:

```
<restricted table name list> ::= <restricted table name> [, ...]
<restricted table name> ::= <table name> [WHERE <restriction clause>]
<table attributes list> ::= <table attributes> [, ...]
<table attributes> ::= <table name>(<column name list>)
```

restricted table name list in the context of specialization is also known as a ertable list. The following rules give the semantics for specialization:

) Each table name in the supertable list is the name of a table, not a view or domain. Every column of every supertable is inherited by the subtable.
) The graph of the table inheritance hierarchy is a directed acyclic graph (DAG), i.e. there are no cycles; thus a table cannot be a direct or indirect subtable of itself.
) If tables in the supertable list have a common predecessor (i.e. they transitively inherit from a common root table), then there will be *one* copy of the shared columns. If these shared columns have different names and there is no AS clause, then the inherited column retains the name used in the leftmost table of the supertable list.
) If an AS clause is specified, then the columns inherited from the supertables are renamed to the names in the AS clause. The number of names in the AS clause must be the same as the number of columns being inherited by the subtable.

(5) If the AS clause is not specified, then the name of each inherited column name must be unique among all columns inherited by the subtable being defined and among all column extensions, if any. Note that if conflicting columns come from the same predecessor, rule 3 will still be valid since there will be only one copy of the shared columns in the subtable.

(6) If CORRESPOND is specified, the arity of each set of table attributes in the table attributes list must be the same. The nth column (or attribute) of each set of table attributes are known as *corresponding columns*. The corresponding columns must be of compatible types, and will become one column in the subtable.

(7) The order of the columns in the subtable being created is:
   (a) the corresponding columns (rule 6), if any
   (b) the rest of the inherited columns in the order they appear in each supertable, starting with the leftmost table in the supertable list (subject to rule 3)
   (c) the additional column extensions, if any

(8) The inheritance hierarchy has *set inclusion* semantics. Formally, this means if a table T1 is a subtable of table T2, then the set of all rows in T2 will contain all rows of T2 AND all the rows of T1. This implies:
   (a) on any insertion, deletion, or modification of a set of tuples in a subtable, the subtuples formed by projecting on inherited attributes are (logically) inserted, deleted, or modified in the supertable(s).
   (b) On any deletion or modification of a set of tuples in a supertable, the corresponding tuples are deleted or modified in all subtables.

(9) For any supertable with a WHERE clause specified, the elements of the supertable which satisfy the WHERE clause are *also* elements of the subtable. This is another consequence of set inclusion semantics, and implies that when inserting a tuple into a supertable, that tuple is inserted into any subtables having a WHERE clause which that tuple satisfies.

This syntax follows an ANSI SQL3 proposal in its basic structure. Both SQL3 and Intelligent SQL include the supertable list and the AS clause. The SQL3 proposal does not, however, include the WHERE clause (vertical modification) or the CORRESPOND clause. The SQL3 proposal also allows a "LIKE <tablename>" clause as part of the table elements, which seems redundant and clumsy; a user might combine use of a supertable list and a LIKE clause:

```
CREATE TABLE StudentStaff
        SPECIALIZES Student
        (GrantNo      INTEGER,
        OfficeMate    CHAR(20),
        LIKE Staff)
```

instead of using

```
CREATE TABLE StudentStaff
        SPECIALIZES Student, Staff
        (GrantNo      INTEGER,
        OfficeMate    CHAR(20))
```

The LIKE clause will also be confusing to many users who are familiar with one of the many packages that uses LIKE as a search condition for character strings.

As in SQL3, the AS clause is used to rename the inherited columns in the subtable. It is most useful in cases of multiple inheritance where two of the supertables have different names for a corresponding column. The CORRESPOND clause is needed when a subtable inherits from two supertables whose common columns are in different orders, especially if those two supertables do not have a common ancestor. Consider, for example,

```
CREATE TABLE Employees (
                EmpName     CHAR(20),
                Salary      INTEGER,
                Department  INTEGER,
                EmpAge      INTEGER)

CREATE TABLE Students  (
                StudentName CHAR(20),
                StudentAge  INTEGER,
                GPA         FLOAT)


CREATE TABLE ResearchAssistants (
        SPECIALIZES Employees, Students
        CORRESPOND  Employees(EmpName, EmpAge)
                    Students (StudentName, StudentAge)
        AS Name, Age, Salary, Dept, GPA
        (ResearchArea       CHAR(20)
```

In this example, EmpAge and StudentAge are common columns, but EmpAge is the fourth column of Employees, while StudentAge is the second column of Students. Without the CORRESPOND clause, Salary might correspond to StudentAge and the semantics for the subtable would not be correct. Or Salary, StudentAge, and

EmpAge would all become distinct columns, and the semantics for the subtable semantics would not be correct. The CORRESPOND clause ensures that the age columns will be related correctly and made into one column in ResearchAssistants.

As stated in rule 8, table inheritance in Intelligent SQL has set inclusion semantics. In the first example above, the table Persons includes all tuples from Staff and Student, i.e. the query "SELECT * FROM Persons" will return all tuples from Persons, Staff, AND Students (see Figure 2). Thus, inserting a tuple into either Staff or Students also affects Persons. Similarly, deleting or updating tuples in Persons affects tuples in Staff and Students. Inserting tuples into Persons, however, affects only the Persons table, since neither Staff nor Students was specialized with a WHERE clause restriction on Person. But, if Students had been created as:

```
CREATE TABLE Students
        SPECIALIZES Persons WHERE Age <= 18
```

then inserting a tuple into Persons with a value of 17 for Age would also (logically) insert a tuple into Students.

Set inclusion semantics has interesting implications for subtables created with WHERE clause restrictions. Two subtables created from the same supertable could have non-empty intersections. Look at two subtables of Employees (Figure 3):

```
CREATE TABLE HighlyPaidEmployees
        SPECIALIZES Employees WHERE Salary > 50,000

CREATE TABLE AveragePaidEmployees
        SPECIALIZES Employees WHERE Salary > 25,000
                                    and Salary < 75,000
```

If we insert the tuple ("John Smith", 60000, "Toys", 40) into Employees, that tuple automatically becomes part of BOTH HighlyPaidEmployees and AveragePaid-Employees. A SELECT * query from either of those subtables would return the tuple for John Smith.

Generalization

Specialization uses a top-down approach to database construction or definition. Generalization is the complement to specialization; it uses a bottom-up approach. In

generalization, supertables are constructed from existing tables. These supertables can then be used for creating new specialized tables.

Generalization allows the creation of supertables through the projection and union of existing tables, using the following syntax:

CREATE TABLE <table name>
GENERALIZES <restricted table name list>
[CORRESPOND <table attributes list>]
[AS <column name list>]

where (the following specifications are the identical to the ones above for specialization):

<restricted table name list> ::= <restricted table name> [, ...]
<restricted table name> ::= <table name> [WHERE <restriction clause>]
<table attributes list> ::= <table attributes> [, ...]
<table attributes> ::= <table name>(<column name list>)

The restricted table name list in the context of generalization is also known as a subtable list. The following rules give the semantics for generalization:

(1) Each table name in the subtable list is the name of a table, not a view or domain.
(2) The graph of the table inheritance hierarchy is a directed acyclic graph (DAG), i.e. there are no cycles.
(3) If CORRESPOND is specified, then the arity of the supertable being created is equal to the arity indicated in this clause (i.e. the number of columns in the supertable is the number of columns in the subtables that correspond). The arity of each set of table attributes in the table attributes list must be the same. The nth column (or attribute) of each set of table attributes are known as *corresponding columns*. The corresponding columns must be of compatible types, and will become one column in the supertable.
(4) Without a CORRESPOND clause, the tables in the subtable list must all have the same arity. In this case, the nth column of each table correspond to each other and are known as *corresponding columns*. Again, the corresponding columns must be of compatible types, and will become one column in the supertable.

(5) The AS clause is used to rename the columns of the supertable that is being defined; the number of names in the AS clause must be the same as the number of corresponding columns. Without the AS clause the names of corresponding columns in each subtable must be the same.

(6) The new supertable will be instantiated with the values from the subtables projected on the corresponding columns.
This is a consequence of the set inclusion semantics for table hierarchies (see rule 8 of specialization).

(7) For any subtable with a WHERE clause specified, the elements of the supertable which satisfy the WHERE clause are also elements of the subtable. This is a consequence of the set inclusion semantics for table hierarchies (see rule 8 for specialization). This implies that when inserting a tuple into a supertable, the tuple is inserted into any subtables having a WHERE clause which that tuple satisfies.

Note that rules 1, 2, 5, 7, and 8 are the same as rules for specialization. This makes sense, as generalization and specialization are two sides of the same coin. The semantics for a supertable-subtable pair created through generalization are no different than the semantics for a supertable-subtable pair created through specialization. Also note that the SQL3 proposal has no notion of generalization. Intelligent SQL is more complete than SQL3 in allowing both top-down and bottom-up definition of the database schema. Generalization adds flexibility, which is important in the commercial arena where a database administrator may have to deal with existing tables and may not have the luxury of defining a table hierarchy from scratch.

For example, if there are existing Employee and Student tables, and there is a need for a Persons table that contains the common parts of both of these tables, the Persons table can be created by:

```
CREATE TABLE Persons
        GENERALIZES Employees, Students
        CORRESPOND  Employees(EmpName, EmpAge)
                    Students (StudentName, StudentAge)
        AS Name, Age
```

where Employees and Students are again defined as:

```
CREATE TABLE Employees (
        EmpName     CHAR(20),
        Salary      INTEGER,
        Department  INTEGER,
        EmpAge      INTEGER)

CREATE TABLE Students  (
        StudentName CHAR(20),
        StudentAge  INTEGER,
        GPA         FLOAT)
```

Note that generalization, like specialization, has set inclusion semantics, and these affect the WHERE clause option.

When no CORRESPOND clause is used (rule 4), the supertable is in effect a union of the subtables, with no projection of a subset of the subtables' columns. This is very useful in distributed databases for creating a global schema from several existing local database schemas, as shall be described below.

Assign

When there are related tables with related tuples, tuples may need to move from one table to another. For example, a Person may be hired and become a member of Staff. Intelligent SQL includes an ASSIGN operator to move tuples back and forth between supertables and subtables. When moving from a subtable to a supertable, any additional columns that are not in the supertable are dropped. When moving from a supertable to a subtable, values for additional columns are specified similar to an insert statement. The syntax for the ASSIGN statement is:

```
ASSIGN INTO <table 1> [(<column list>)]
        FROM <table 2>
        WHERE <search condition>
        [VALUES(<value list>)]
```

Thus, to make John Smith from the Persons table a member of Staff:

```
ASSIGN INTO Staff(Salary, OfficeNo)
        FROM Persons
        WHERE Name = "John Smith"
        VALUES(40000, "AX700")
```

To fire everyone with a salary over 100,000:

```
ASSIGN INTO Persons
    FROM Staff
    WHERE Salary > 100,000
```

Other Data Definition Constructs

Specialization and generalization interact with many other data definition constructs in SQL. Some of these interactions can be quite complex. This section will discuss the following constructs and actions:

(1) Primary keys
(2) Foreign keys
(3) Unique columns
(4) DROP/CREATE INDEX
(5) DROP TABLE

Each table can have at most *one* primary key. In an inheritance hierarchy, a primary key is either

(a) defined with the table definition. If the table being defined is a generalization, then defining a primary key invalidates the primary key definitions of all its subtables. If a primary key is defined in a supertable, it is an error to define a primary key for a subtable.
(b) inherited from a supertable.

Note that two sibling tables in an inheritance hierarchy can each have their own primary key, as long as their parent table does not have a primary key defined. If the definition of a supertable is altered to add a primary key, the primary keys of all of its subtables will be invalidated. A primary key can be dropped only by the most general table containing the key.

Foreign keys behave in a similar manner. When a foreign key constraint is defined for a supertable, it is automatically inherited by all of its subtables. But it is an error for a subtable to define or add a foreign key constraint involving inherited attributes (columns). Like primary keys, a foreign key can only be dropped by the most general table containing the foreign key.

Any table in an inheritance hierarchy can define a unique constraint on a column or set of columns. It is an error, however, for a subtable to define or add a unique constraint involving inherited attributes. Again, a unique constraint can only be dropped by the most general table containing the unique constraint.

When an index is created on supertable, it applies to all tables below it in the inheritance hierarchy. If an index of the same name already exists on one of the subtables, the create will fail. The index can be dropped only by the most general table containing it.

In general, dropping a supertable does not remove its subtables. The DROP TABLE statement has an option to drop the entire subtree,

DROP TABLE [INCLUDING SUBTABLES] <table name>

but the default behavior is to only remove the table specified. This behavior follows the principle of defaulting to the least destructive action possible, and requiring the user to explicitly specify any further action desired. Otherwise, a user, not aware that a table has subtable, could accidentally delete a subtree of dozens of tables with a single command. When just the supertable is dropped, its subtables will continue to exist, and all constraints will be propagated to the subtables. The subtables will also become direct subtables the dropped table's supertable (see Figure 5).

Distributed Databases

Specialization and generalization are important object oriented concepts. They allow users to build new tables on top of other tables. An important application of these concepts is in the integration of distributed databases. In these cases, the tables being specialized or generalized are actually stored on different nodes of a distributed database environment. For instance, assume that a company has two plants, one in Los Angeles and one in San Diego, and has their corporate headquarters in Las Vegas. They have separate databases at each site, each with their own employees table: LAEmployees, LVEmployees, and SDEmployees (see Figure 4a). They now want to be able to query against a combined employee table and have acquired the technology to link all three databases together in one combined database (another feature of Intelligent SQL). They can create the table AllEmployees as follows:

```
CREATE TABLE AllEmployees
    GENERALIZES
        LAEmployees  WHERE Location = "Los Angeles",
        SDEmployees WHERE Location = "San Diego",
        LVEmployees  WHERE Location = "Las Vegas"
```

where Location is the site location for each employee, and is defined in the database for each site (see Figure 4b). This table (and any generalized table) can be later

used as a supertable for specialization. If the company were to open an office in San Francisco with its own database, they could create a table SFEmployees from AllEmployees (see Figure 4c):

```
CREATE TABLE SFEmployees
        SPECIALIZES AllEmployees WHERE Location = "San Francisco"
```

Conclusion

Intelligent SQL has a consistent definition of specialization and generalization as two complementary pieces of inheritance. Intelligent SQL would make a good basis for a standardization of inheritance, especially since generalization is not addressed at all in the ANSI SQL3 proposal. Intelligent SQL also has several useful additions to the functionality described in SQL3, including a CORRESPOND clause and a WHERE clause. The CORRESPOND clause allows users to specify which columns are to be inherited. The WHERE clause aids in administration and transparency for distributed databases, and also allows tupies to be common to more than one subtable. Thus, Intelligent SQL provides the flexibility in inheritance that commercial applications need.

REFERENCES

Cox, B., "Message/object programming: An Evolutionary Change in Programming Technology," *IEEE Software*, January 1984, pp. 50-61.

Kuper, G.M., and Vardi, M.Y., "On the Expressive Power of the Logic Data Model," *Proceedings of ACM SIGMOD*, 1985, pp. 180-187 Austin, Texas.

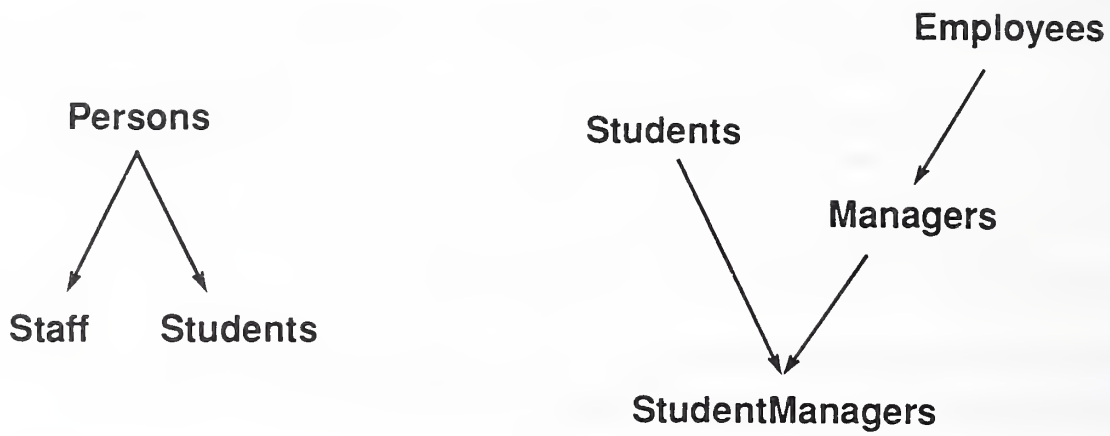
Khoshafian, S., Parsaye, K.,, and Wong, H.K.T., "Intelligent Database Engines," *Database Programming and Design*, July 1990.

Khoshafian, S. "Intelligent SQL", *Proceedings of OODBTG*, May 1990.

Khoshafian, S. and Abnous, R. *Object Orientation*, John Wiley & Sons, Inc., July, 1990.

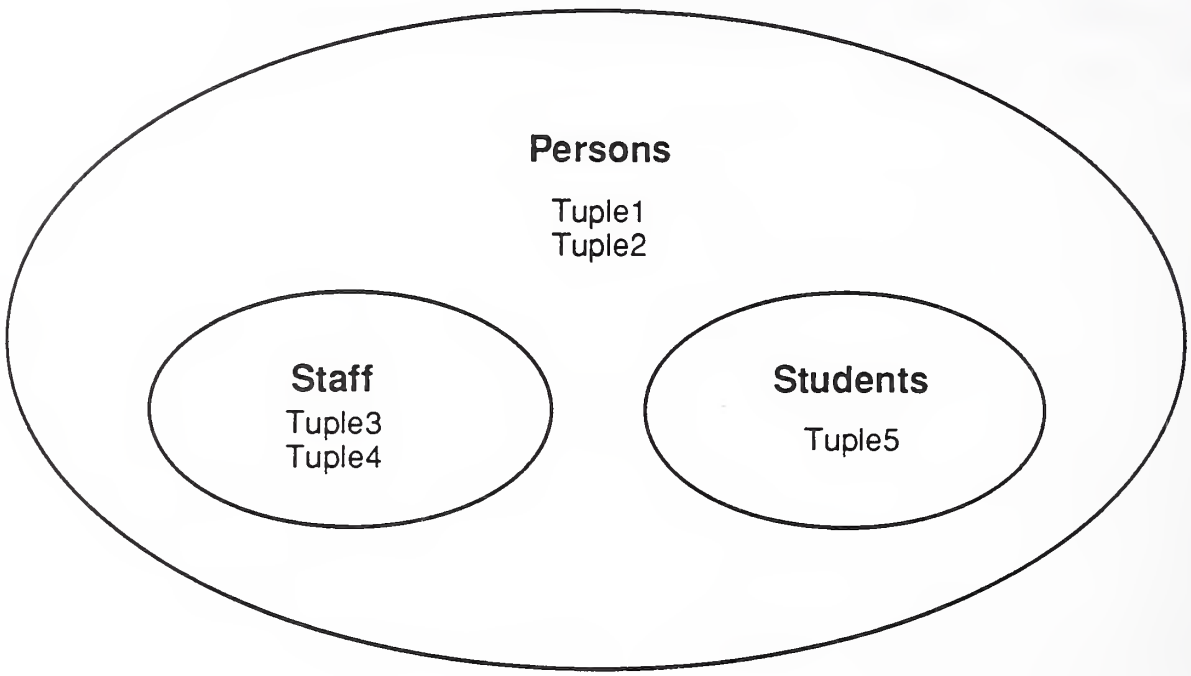
Shipman, J., "The Functional Data Model and the Data Language DAPLEX", *ACM Transaction on Database Systems*, vol. 6, no. 1, 1981

Tsur, S., and Zaniolo, C., "LDL: a Logic Based Data Language," *Proceedings of VLDB*, August 1986, Kyoto, Japan.



(a) Simple inheritance (b) Multiple inheritance

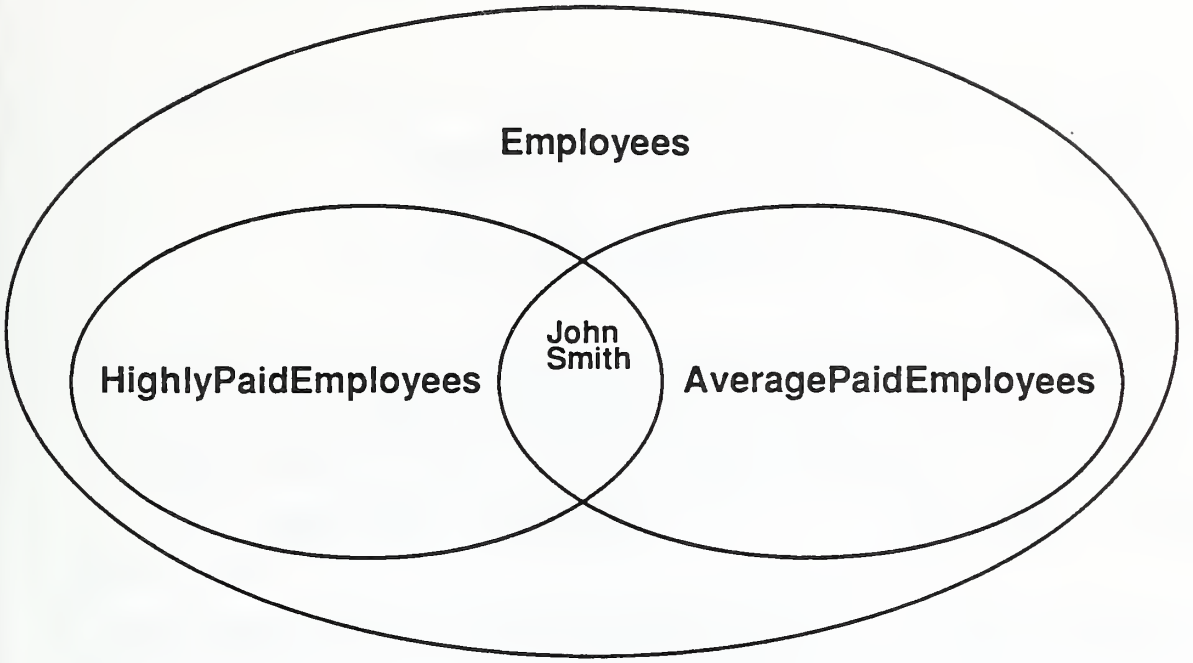
Figure 1



SELECT * FROM Persons *returns* { Tuple1, Tuple2, Tuple3, Tuple4, Tuple5 }

Set Inclusion semantics of inheritance

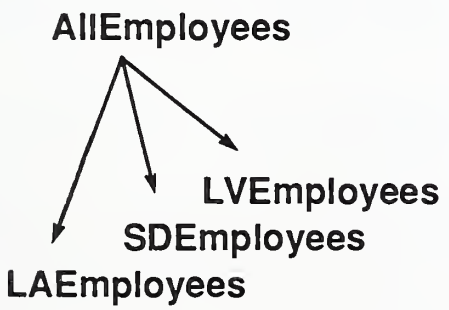
Figure 2



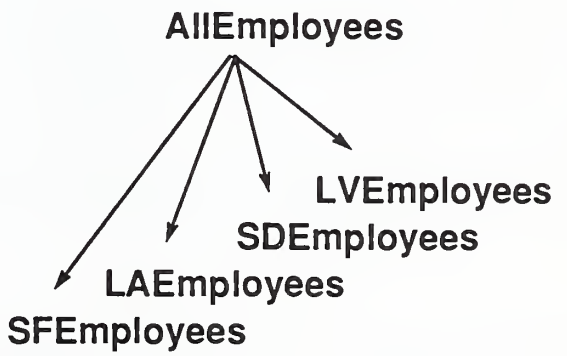
Set Inclusion with Intersection of subtables

Figure 3.

(a) Unrelated distributed tables

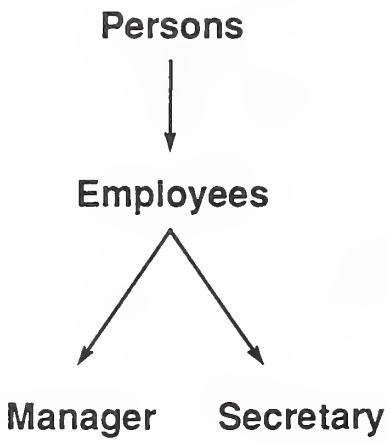


(b) Related tables after Generalization

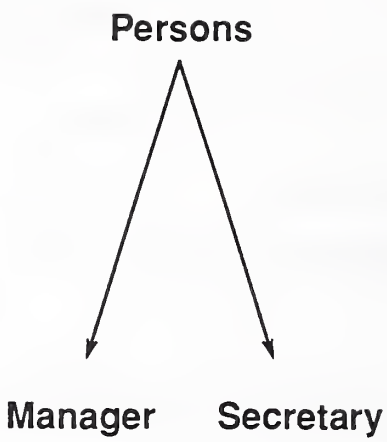


(c) Add new table with Specialization

Figure 4



(a) Initial hierarchy



(b) Hierarchy after dropping Employees

Figure 5

Object Databases as Generalizations of Relational Databases

David Beech and Çetin Özbütün

Oracle Corporation
500, Oracle Parkway, Redwood Shores, CA 94065

ABSTRACT: One attractive approach to object databases is to see them as potentially an evolutionary development from relational databases. This paper concentrates on substantiating the technical basis for this claim, and illustrates it in some detail with an upwards-compatible extension of ANSI SQL2 for conventional objects. This could serve as a foundation for the development of higher-level facilities for more complex objects.

1. Introduction

Object technology offers a great opportunity to bring together developments in programming languages, open systems, and database systems. We foresee the need for object database (ODB) systems which can support a variety of language interfaces (e.g. C++, Smalltalk, Eiffel, COBOL with objects!), and participate in open systems data interchange (e.g. via the abstract syntax notation standard, ASN.1). Users (and builders) of today's relational systems would also be grateful if such systems could grow in a non-disruptive way by extension of the relational model.

We shall show here that such a generalization is indeed possible. An analogy exists with the way in which C++ has been developed as a compatible extension of C. Although, as we shall describe later, the details of the generalization differ, the principle remains the same, that there is much to be gained by exploiting similarities between object models and their predecessors. In fact, SQL provides in many ways an easier starting point than C, because of the essential simplicity of the relational model. Of course, an exercise of this kind must be judged by the quality of its results. We shall try to show that although all languages are imperfect (and SQL is no exception), a suitably extended SQL can stand comparison with other object languages. It also benefits from carrying along with it facilities for set-oriented operations and queries, constraints, authorization, and so forth, which are not usually found in programming languages. On this foundation, one can build the functionality of complex and distributed objects and object views, version control, management of long transactions, etc. Within the confines of this paper, we shall not attempt to describe these superstructures, since such design work can in any case be quite similar across different languages. We limit our goal to establishing that an extended SQL can provide an effective language foundation for such developments, in addition to offering a smooth migration for today's users (direct or indirect) of SQL.

An object database management system (ODBMS) could then contain within it all the functionality of a relational database management system (RDBMS). A database could contain general classes and object instances, as well as the relational special case of classes represented by tables, where the object instances are simple tuples. Such an ODBMS could be built from the ground up, but it also appears to be quite feasible to extend the implementation of an existing RDBMS to give high performance for long transactions with complex objects. There is naturally more involved than cosmetic changes at the SQL interface to the system, and deeper optimizations must be made, but a great deal of the existing functionality can be utilized, and its equivalent must in any case be provided to support the relational subset and the operational characteristics of a production system.

Further discussion of these and other aspects of the future development of relational systems may be found in the "Third-Generation Database System Manifesto" [STON90].

2. Main Concepts

Assuming that the reader is somewhat familiar with object technology, we give here a brief summary of the conceptual model and terminology to be used in what follows.

Object Databases

An *object database* is designed to contain a collection of objects --- not, of course, physical objects, but information objects. We will first try to give an intuitive idea of what these objects are, and then give a more technical description of their realization as an extension of the relational model.

Objects

An *object* in the database is a time-dependent collection of information which may often be used to model some object outside the database --- a physical object such as a person or a manufacturing plant, an abstract or hypothetical object such as a design or a plan, a scientific entity or concept such as a gene, or a fictitious object such as Sherlock Holmes --- in short, anything which might be referred to by a noun or a noun phrase.

Identity

By saying that the information in the object is time-dependent, we allow that it can change, and yet the object can still be considered the same object. This is the property of *identity* --- the property which distinguishes an object from any other object, regardless of its information content at a given moment.

Attributes

The information content of an object is given by its *attributes*. For a person, these attributes might include name and address, date of birth, father, mother, color of hair, phone numbers, etc. (see Fig. 1). In general, the value of each attribute such as father or mother is itself an object --- or more strictly, a reference to an object, which can thus exist independently and be shared between many attributes --- multiple person objects can refer to the same father and mother. It is a question of database design as to whether the values of attributes such as address and date are modeled as references to address and date objects (which might carry additional information about their addresses and dates), or are represented as simple character string and date values within the defining object. Even this last alternative may be thought of as a degenerate case of a reference to an object, where the referenced object happens to represent an immutable value, and can therefore be copied into each attribute which would have referenced it.

Behavior

So far, we have been considering the structural aspects of information objects, but this is only half the story. Before developing these aspects further, we need to examine the other half of the conceptual model, concerned with the behavior of objects. This behavior may be categorized in three ways.

First, it should be possible to specify *operations* to be performed on the database. These may be either general-purpose data manipulation language (DML) operations, or user-defined *routines*

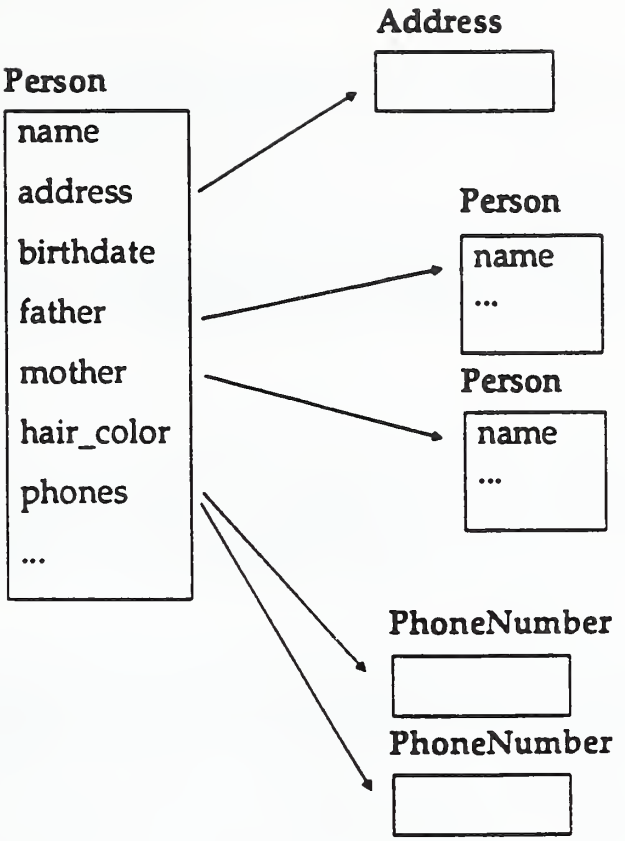


Figure 1
Object Attributes

(functions and procedures). The DML operations allow objects to be inserted and deleted, updated, and queried. User-defined routines may be stored and applied to objects in the database to achieve higher-level behavior definable in terms of the DML primitives.

Second, non-procedural *constraints* may be specified to inhibit the behavior of objects. For example, the birthdate of a person object in a database could be constrained to lie within certain limits.

Third, *triggers* may be specified which will cause a certain operation to be performed whenever a given situation arises, such as the violation of a constraint, or the occurrence of a particular kind of operation.

Classes

The structural and behavioral aspects of the model are brought together in the concept of a *class*. Similar objects may be grouped together in a class, and an object is said to be an *instance* of a class

to which it belongs. Thus the objects representing individual people might be instances of a Person class.

In this ODB model, the concept of class incorporates both the meaning or *intension* of the class, and the collection of its instances at any given moment --- the *extension* of the class.

The Intension of a Class

The intension of a class describes the common features of the objects which are instances of the class. Structurally, this includes the specification of what their attributes are. Behaviorally, the class defines the operations, constraints, and triggers applicable to its instances.

The Extension of a Class

The fact that a class has a known extension in this ODB model allows DML operations analogous to those of the relational data model to be provided. The extension of a class is a set of objects on which operations may be performed to retrieve or manipulate sets of objects in a single operation.

Class Inheritance

A class may be defined to be a *subclass* of one or more other classes. This is intended to model the intuitive notion of a classification hierarchy, in which a subclass is a specialization of its *superclasses*, having all their properties and possibly some additional ones of its own. For example, Employee and Student classes may each be a subclass of the Person class. The subclass is said to *inherit* the properties (attributes and operations) of the superclasses. A person who is both an employee and a student might be modeled by a class EmployeeStudent which is a subclass of both Employee and Student, and inherits the properties of both by *multiple inheritance*.

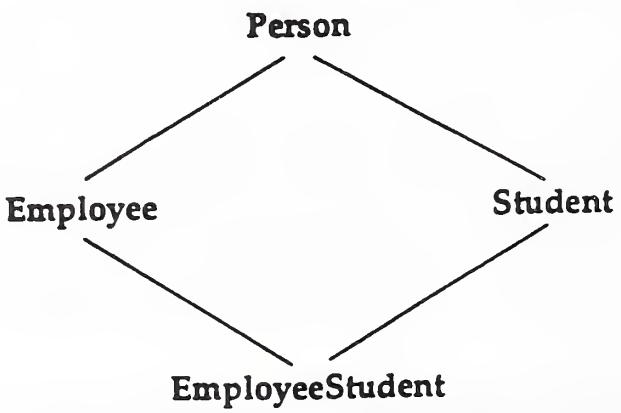


Figure 2
Multiple Inheritance

Dynamic Specialization

An important feature of the inheritance of operations is known as *dynamic specialization*. The specialization occurs when an operation with a given name defined in a certain class C1 is redefined in some direct or indirect subclass C2. For instances of C2 and its subclasses, the redefined operation is used or inherited in preference to that defined in C1, allowing more specialized semantics to be provided for these special kinds of C1. The specialization is dynamic, since the decision which operation to invoke depends on the most specialized class of the actual operand at run-time, not on its declaration which might merely constrain it to be a C1. Thus the same code can cause different operations to be performed on different operands, and is robust against the definition of additional subclasses and specialized operations.

Authorization and Encapsulation

There need to be two ways of controlling access to information in an object database of the kind described so far.

First, there should be facilities for controlling which users of the system are allowed to see and manipulate which information. The creator of each object is its *owner*, and may grant certain privileges on the object to other users. Privileges may also be revoked later.

Second, parts of objects of a given class may be *encapsulated* so that they are accessible only within the bodies of routines associated with that class. This is part of the software engineering discipline of the use of objects to construct large modular systems, and is orthogonal to the user authorization just described --- the owner of such a routine needs to grant to other users the privilege of using it.

3. The Basic Principle

The fundamental underlying reason for being able to evolve from a relational database to an object database of the kind outlined above is that a row in a table can be thought of as representing a simple object, whose identity is defined by a primary key and whose class is the table to which it belongs. In fact, declarative referential integrity in relational SQL now encourages the user to think in this way, with the REFERENCES clause for the foreign key indicating a reference to another "object".

To illustrate this in more detail, let us consider the ominously familiar relational example:

```
Create table Department
        (deptno integer primary key,
         name char(20));

Create table Employee
        (empno integer not null unique,
         name char(30),
         dept integer references Department);
```

A CREATE TABLE statement defines both a data structure and a collection of instances of that structure. The data structure is a normalized tuple, and the collection is a multi-set (or a set, when duplicate rows have been disallowed by a UNIQUE constraint across all the columns) of such tuples. The "REFERENCES Department" clause is a referential integrity constraint, which

requires that any integer in the dept column must be equal to some existing primary key in a row of the Department table.

Compare this with what we will postulate as comparable class definitions:

```
Create class Department
        (name char(20));

Create class Employee
        (empno integer not null unique,
         name char(30),
         dept references Department);
```

The CREATE CLASS statement can similarly define both a data structure, and a collection of object instances that have that structure. Assuming that deptno was artificially generated to serve as a primary key in the relational case, we can omit it here, and allow the system to generate an implicit object identifier (oid) when an object is created. The dept attribute of an Employee object must match the oid of some Department, and no other datatype is specified for this attribute --- "REFERENCES Department" has become the datatype, as with a constrained pointer type in a programming language (cf. "REF Department" in Simula, or "Department *" in C++).

The class definitions above, which merely replace the TABLE keyword by CLASS in the SQL syntax, already resemble simple class declarations in a programming language. (They also carry along useful features such as various kinds of constraints.) By adding a class hierarchy, methods, encapsulation, and a richer type system, one can arrive at a complete object model in which the relational model survives as a special case. If the type system allows not only struct, but array, sequence, set, and choice constructors to be applied recursively to an adequate range of primitive types, this will match the ASN.1 descriptive capability, and will exceed that of most programming languages by handling sets.

4. Objects in SQL

We have already seen some simple class definitions, so we can now introduce some richer data types by using the set constructor, in combination first with a scalar type and then with a reference type (where we allow **references** to be abbreviated to **ref**:

```
Create class Person
        (name char(30),
         phones set of integer,
         children set of ref Person);
```

Classes can also be defined in a classification hierarchy:

```
Create class Employee subclass of Person
        (empno integer,
         manager ref Employee);

Create class Student subclass of Person
        (courses set of ref Course );

Create class EmployeeStudent subclass of Employee, Student
        ( );
```

A subclass inherits attributes from its superclasses, and can define additional attributes of its own, although it does not need to.

A further generalization can then allow procedures and functions (often known as *methods*) to be associated with class definitions. A scheme for *encapsulation* of private parts of an object's data structure would permit access only by appropriate procedures and functions. We will not pursue these aspects in this paper, but one speculative syntax could be:

```
Create class Person
        (name char(30),
         phones set of integer,
         children set of ref Person,

         function age(ref Person) return integer
                 begin /* flattering computation */ end,

        private:
         birthdate date );
```

Other new Data Definition Language (DDL) statements can allow the user to Alter Class and Drop Class. Many existing DDL statements such as Create Index, Rename, and Grant and Revoke privileges, are applicable to classes also.

Turning to the Data Manipulation Language (DML) statements of SQL, new object instances may be inserted into a class using the same syntax as for inserting rows into a table:

```
Insert into Employee (name)
        values('J.Doe');
```

An Employee object has been created, with its "name" attribute (inherited from the Person class) initialized with the specified value, and its other attributes null.

Other existing DML statements which can be made applicable to classes and objects are Select, Update and Delete, Commit, Rollback and Savepoint.

We could perform an update on the Employee object above:

```
Update Employee
        insert into phones values {3581234, 5068497}
where name='J.Doe';
```

Nested insert, update and delete clauses are provided to operate on set-valued attributes, and set expressions may be enclosed in braces.

A simple query will show the result of this:

```
Select name, phones from Employee;

NAME        PHONES
J.Doe       3581234
            5068497
            ..........
```

The system generated a unique object identifier when it created the Employee object, and this could be retrieved if desired by specifying "oid" in a query as though it were an attribute name. Rather than dealing with the internal format of an oid, it is usually preferable to retrieve the oid into a *host variable* which can serve as a symbolic way of identifying the object:

```
Select oid into :John
from Employee
where name='J.Doe';
```

In fact, we could have bound a host variable to the object when we created it:

```
Insert into Employee (name) alias :John
        values('J.Doe');
```

The name of the host variable is chosen by the user, and the preceding : is an SQL convention for a variable name from a surrounding environment. If these examples were being entered interactively, the host variables would be part of the session environment, and would be valid only for the life of the session.

Once bound, a host variable could be used to refer to an object directly, e.g. the Update statement above could have been expressed:

```
Update Employee instance :John
        Insert into phones values {3581234, 5068497};
```

Relationships between objects in the database are often expressed by attributes whose values are references to other objects, e.g.:

```
Insert into Employee (name) alias :Hal
        values ('H.Snooks');

Update Employee instance :John
        set manager = :Hal;
```

When retrieving the manager of an Employee, we probably wish to see some attributes of the referenced object rather than merely its oid, and function notation can be provided as a simple way of achieving this without having to write out an explicit join condition:

```
Select name, name(manager) as manager
from Employee;
```

NAME	MANAGER
J.Doe	H.Snooks
H.Snooks	

Queries of this kind are still returning values rather than objects, but it would lead us beyond the scope of the present paper to discuss a SELECT OBJECT form of query which could return objects (including aggregates of objects connected by attributes with REF datatypes).

5. Relationship to Other Object Models

The above examples emphasized the evolutionary possibilities of an ODBMS as an extension of an RDBMS, and they would have been easiest to follow for the reader already familiar with SQL. However, an important design goal for an object database should be to recognize the merits of object concepts as they have arisen in other spheres, and to encourage convergence between the data models used in programming languages, open systems, CASE tools, and database systems, with a view to better integration of the information systems of the future. Underlying this technical level of object modeling, there is of course the hope that such a model comes closer to the intuitive concepts of objects employed by human users. Even though relatively few end users will actually deal with systems at the technical level, the task of providing problem-oriented user interfaces and performance should be greatly simplified.

Therefore we will provide some perspective here for the reader who may be approaching from the direction of programming languages, or open systems, or entity-relationship design tools.

5.1 Programming Languages

The main outline of declaring classes in a hierarchy, of being able to create objects as instances of those classes and refer to them symbolically with variable names, and of using an object reference to relate one object to another, is widespread in programming languages. By comparison with this generic programming language model, the object database model suggested here offers three main extensions.

The first extension is that a class is regarded as defining not only a template for its instances, but also a collection of those instances that will be automatically maintained by the system. This goes hand in hand with the ability, expected of a DBMS, to perform queries, updates and deletions over such collections. An important corollary of this is that relationships between objects may be processed by specifying conditions on collections of them, rather than by stepwise navigation between individual objects, and many-to-many relationships may be treated symmetrically and conveniently as collections of pairs of object references.

With a strong treatment of collections of objects in the DBMS, it is natural for the ODB to have a second extension compared to most programming languages by offering direct support for set-valued attributes, as illustrated earlier.

Third, the ODB model includes declarative support for integrity constraints on objects. The values of attributes can be constrained to satisfy specified conditions, and referential integrity can be enforced - i.e., an object cannot be deleted while there are any references to it from other objects or tables in the database.

5.2 Open Systems

The Open Systems Interconnection standard for data interchange, the so-called abstract syntax notation ASN.1 (ISO 8824), is based on a partial object model. The structure of a data object may be composed out of simple scalar data types by the recursive use of record, array, sequence, set, and choice constructors. An object database should support this richness in its type system.

ASN.1 does not support a class hierarchy, nor does it define inter-object references. However, proposals now being considered by ISO would add both references and functions to the standard, increasing the similarity of scope with the programming language and ODB models.

5.3 Entity-Relationship Models

Entity-relationship (ER) models are often used in database design to capture intuitive notions of object structure. Rather than requiring that entity sets be mapped to relational tables, the ODB model makes it possible to map them to object classes. (Relationship sets could still be mapped to tables.)

The object concept is more general than the original ER concept of an entity, allowing for richer data structure, a class hierarchy, and (especially) functions and procedures to define the semantic behavior of objects. The general consistency of the ER and object approaches is illustrated by the existence of extended ER models which add features such as an entity type hierarchy.

Conclusion

Object technology holds great promise for the coming decade, and it is feasible for relational database systems to be generalized and optimized to support the concepts via graceful evolution.

References

[STON90] M. Stonebraker, L.A. Rowe, B. Lindsay, J. Gray, M. Carey, M. Brodie, P. Bernstein, D. Beech: Third-Generation Database System Manifesto. Proceedings of the IFIP DS-4 Conference on Object-oriented Databases, Windermere, England. July 1990. Reprinted in ACM SIGMOD Record, Vol 19 No 3, Sept 1990.

Appendix: SQL Syntax

In this appendix, we specify some possible syntactic extensions to SQL for defining and manipulating classes and objects. Since most of the ANSI SQL syntax applicable to tables is also applicable to classes, we do not repeat the ANSI SQL rules; instead we give the syntax only for new statements and for those statements that are expanded to fully utilize object concepts. Notice that the new statements are also modeled on existing statements, changing the keyword TABLE to CLASS.

We have used the draft ANSI/ISO SQL2 manual (X3H2-90-264) as the starting point. All categories shown within double angled brackets (e.g. << column name >>) are defined in that document. There are a few debatable points about some syntactic details, and the exact expression of the rules in the partially context-dependent style of the ANSI document would require the assistance of some X3H2 wizards, but the intent should be clear. Of course, these "format specifications" would need to be supplemented by "syntax rules" and "general rules" in the style of the ANSI document to provide a complete specification, but this is more than needed here to indicate the rather straightforward nature of the extensions.

==========

5.5 Names

Function

Specify names.

Format

<column name> ::= [<table name> ::] <<column name>>

Notes

1. For syntactic convenience, we extend the meaning of <column name> to cover attribute names also, and <table name> to cover class names. An attribute name may optionally be qualified by a prefixed class name (and a :: separator, as in C++) to resolve ambiguities. The prefix class is where the attribute is inherited from (i.e. the attribute is defined in that class, or inherited by it).

==========

5.6 <data type>

Function

Specify a data type.

Format

<data type> ::=
<<data type>>
| <reference data type>
| <struct data type>
| <array data type>
| <set data type>
| <sequence data type>
| <choice data type>
| <type name>

<reference data type> ::=
{ REFERENCES | REF } <table name>

<struct data type> ::=
STRUCT (<column name> <data type>
[{, <column name> <data type> } ...])

<array data type> ::=
ARRAY [(<size>)] OF <data type>

<set data type> ::=
SET [(<size>)] OF <data type>

<sequence data type> ::=
SEQUENCE [(<size>)] OF <data type>

<choice data type> ::=
CHOICE (<column name> <data type>
[{, <column name> <data type> } ...])

<type name> ::= <<identifier>>

<size> ::= <<unsigned integer>>

Notes

1. We add the above data type constructors to the SQL2 data types. They cover the composite types defined by most programming languages and by the ASN.1 standard.

==========

5.8 <column reference>

Function

Reference a named column or attribute.

Format

<column reference> ::=
[<<qualifier>> .] <general reference>

<general reference> ::=
<optionally subscripted reference> [. <general reference>]

<optionally subscripted reference> ::=
<column name> [[<<unsigned integer>>]
[{ [<<unsigned integer>>] } ...]
]

Notes

1. Column reference is extended to allow subscripted references to arrays, and dot qualified references to structs, which may be nested.

==========

5.13 <value expression>

Function

Specify a (possibly composite) value.

Format

```
<value expression> ::=
        <<value expression>>
      | <composite value expression>
      | <function notation>

<composite value expression> ::=
        <set value expression>
      | <array value expression>
      | <struct value expression>
      | <sequence value expression>

<set value expression> ::=
      { <value expression> [ {, <value expression> } ... ] }
      -                                                     -

<array value expression> ::=
      [ <value expression> [ {, <value expression> } ... ] ]
      -                                                   -

<struct value expression> ::=
      ( <value expression> [ {, <value expression> } ... ] )

<sequence value expression> ::=
      [| <value expression> [ {, <value expression> } ... ] |]
      --                                                   --

<function notation> ::=
        <column name> ( <column reference> )
      | <column name> ( <function notation> )
```

Notes

1. Value expressions are extended to allow specifying values for the new composite types.

2. Function notation is introduced as in programming language expressions. For present purposes, its use is limited to using attribute names as functions on references to objects to return the attribute values. This can be used to avoid writing explicit join conditions. Say class A has an attribute a1 which is a reference to class B which has an attribute called b1; in such a

case, one can say b1(a1) to get the b1 attribute from the object "pointed to" by a1. Such functions may be arbitrarily nested.

==========

5.40 <subquery>

Function

Specify a value (possibly composite) derived from a <query expression>.

Format

<subquery> ::=
    <<subquery>>
    | <set valued attribute>
    | <set value expression>

Notes

1. This extension allows that a set valued attribute or expression can occur wherever a subquery can occur. Similar extensions might be made for other composite values.

==========

5.42 <privileges>

Function

Specify privileges.

Format

<action> ::=
    <<action>>
    | SUBCLASS

<grant column list> ::=
    <<grant column list>>
    | <column name>
        [{, <column name> } ...]

Notes

1. The SUBCLASS privilege confers the privilege to create a subclass of a given class. In this way, users have control over who can create subclasses of their classes, and thus can create objects which are also instances of the superclasses.

==========

6.xx <type definition>

Function

Define a type.

Format

<type definition> ::=
CREATE TYPE <type name> AS <data type>

<type name> ::= <<qualified name>>

==========

6.xx <class definition>

Function

Define a class.

Format

<class definition> ::=
CREATE CLASS <table name>
[SUBCLASS OF <table name> [{, <table name> } ...]]
<<table elements>>

==========

7.xx <drop class statement>

Function

Destroy a class.

Format

<drop class statement> ::=
DROP CLASS <table name> [CASCADE]

==========

7.xx <alter class statement>

Function

Change a class and its definition.

Format

<alter class statement> ::=
ALTER CLASS <table name> <<alter table action>>

==========

9.7 <delete statement: searched>

Function

Delete rows of a table or objects of a class.

Format

```
<delete statement: searched> ::=
        <<delete statement: searched>>
        | DELETE FROM <table name>
                [ INSTANCE <<value expression>> ]
                [ WHERE <<search condition>> ]
```

9.8 <insert statement>

Function

Create new objects in a class.

Format

```
<insert statement> ::=
        INSERT INTO <table name> [ <insert attribute list> ]
                [ ALIAS <<target specification>> ]
                [ <<query specification>> ]

<insert attribute list> ::=
        <column name> [ {, <column name> } ... ]
```

9.10 <update statement: searched>

Function

Update rows of a table or objects of a class.

Format

```
<update statement: searched> ::=
        <<update statement: searched>>
        | UPDATE <table name> [ INSTANCE <<value expression>> ]
                <update attribute specification list >
                [ WHERE <<search condition>> ]

<update attribute specification list> ::=
        <update attribute specification>
                [ {, <update attribute specification> } ... ]

<update attribute specification> ::=
          SET <<set clause: searched>> [{,<<set clause: searched>>}...]
        | INSERT INTO <set valued attribute>
                VALUES <set value expression>
        | DELETE FROM <set valued attribute>
                [<set value expression>]
                [ WHERE <<search condition>> ]
        | UPDATE <set valued attribute> <update attribute specification list >
                [ WHERE <<search condition>> ]
```

<set valued attribute> ::= <column name>

<attribute value list> ::=
    <attribute value> [{, <attribute value> } ...]

<attribute value> ::=
    <value expression> | <<null specification>>

Syntax Rules

1. The dangling WHERE clause problem is avoided by attaching a WHERE clause to the nearest possible preceding construct which could have one.

Notes

1. This update statement is extended to handle set valued attributes. One can nest INSERT, DELETE, UPDATE statements within a top level UPDATE statement to insert/delete/update elements into/from/of a set. Since these three operations in SQL are defined to operate on (multi-)sets, it is only natural that they operate on sets at any level, including sets within sets. Similar operations might be provided on other composite values.

==========

A Two Layered Interface Architecture

Dr. Thomas J. Wheeler Judith D. Richardson

US Army CECOM
Center for Software Engineering
Monmouth College - CS Dept.
West Long Branch, NJ 07764

Abstract: The problems of interfacing to external subsystems and using multiple paradigms in a single software system center on resolving the impedance mismatch (the differences in the data models and thought patterns of each paradigm) and how to reflect the differences across the language boundary. One technique is to build an interface between the two paradigms. The interface should strive to resolve the mismatches while providing access to those language features valuable to the problem solution. This means that semantic issues as well as syntactic issues must be addressed. Features of the implementation languages that contribute to the mismatch need to be identified and examined in order to develop a solution. This paper describes a two layer interface architecture based upon the idea of abstract interfaces. This architecture appears to provide a basis for interfacing programming languages to object databases which addresses the semantic gap, or impedance mismatch, between the two as well as the syntactic differences.

Introduction

The method of developing software systems is currently experiencing the largest change since the introduction of general purpose operating systems. Large subsystems conforming to standards are becoming available to be used as the basis for most major software system construction. This increased use of subsystems developed to defacto, national and international standards is forcing the development strategy to become one of composing ("gluing") components together rather than total system development from scratch. The main reason for this standardization trend is the recent leap in the size of software systems and the resulting economic impracticality of developing each system from scratch. This component composition strategy forces developers to address soft-

ware architecture issues early and requires a standard paradigm for interfacing the application specific software to the standard software subsystems.

Another factor influencing the development strategy is the use of multiple paradigms within a single system. Systems are incorporating substantial subsystems based on specialized paradigms, such as, subsystems utilizing artificial intelligence or database principles.

Both of these strategic trends lead to a system structuring pattern of interconnected subsystems with some components written using a modular or object-oriented paradigm and other subsystems written in other languages and using completely different paradigms.

This paper outlines an interfacing architecture designed to enhance these component composition and multiparadigm styles of development. The interfacing architecture is based on a layered approach to interfacing components written in different languages and different styles. The interface bridges the language differences via a syntactic transform and the style differences via a semantic transform. This architecture appears to provide a basis for interfacing programming languages to object databases which addresses the semantic gap, or impedance mismatch, between the two as well as the syntactic differences.

The paper describes a general two layer architecture approach to interfacing and gives some examples of where the approach has been used. This work has grown out of involvement with specifying the architecture to be used in Army systems written in Ada and interfacing to external systems, but the approach is independent of language. The two layer architecture described in this paper has been used as the distributed architecture for the Army Tactical Command and Control System (ATCCS)[CASS 89] as well as the basis for the standard SQL-Ada interface[Wheeler 87] [Richardson, Wheeler and Ferrer 89]. We are currently investigating the issues in interfacing Ada and OODB.

Considerations

Multiple Paradigms :

A paradigm is a model of a particular thought pattern used to solve a class of problems. A paradigm supplies a systematic, cohesive approach to looking at a problem and thereby assists the solver to think more clearly about the possible solutions. The trade-

off is that a paradigm which helps solve one problem may not be flexible enough to solve a wide range of problems [Zave 89]. As a result, large complex systems need to make use of multiple paradigms. Multiparadigm systems are made up of subsystems, each utilizing a different paradigm suited to its particular needs. This allows the overall system to take advantage of the strengths offered by the separate paradigms [Hailpern 86]. The major problems associated with this approach are the mechanism for transferring information and control between the different paradigms and translating between meaningful representations. One approach to solving this problem is to build an interface layer which provides a mapping at the conceptual level of both syntax and semantics.

Data Models :

A data model is a mental model used to abstract the real world into a form that can be used to solve some problem. It contains a representation of real world objects and a mechanism for representing relationships between them as well as a set of operations for their manipulation. Each data model gives a "flavor" to the solutions derived from its use. It is through the use of a data model representation that approaches to problem solving are supported.

The data model for modular languages uses non-persistent, strongly typed, named objects. It can support objects as abstract data types. Access to an object can be direct, through the object's name, or navigational, through access pointers. One of the strengths of this data model is its flexibility; its ability to support a variety of problem solving approaches. The DB interface should not limit the application program to the DB data model, therefore the interface needs the ability to form an abstraction of the services provided by the database.

Mapping Models:

If a one-to-one mapping is not possible between two representations then there is a structure clash [Jackson 75]. To map between the structures, a process for transformation is developed. First the lowest common denominator between the two structures is determined, then algorithms are developed to decompose the first structure and then to build the second structures up from primitives. But, whereas Jackson allows the data structures to be visible to the users, Parnas hides as much of the details as possible [Parnas 72]. Parnas decomposes a system into modules based upon the concept of encapsulating design decisions into modules and then supplies the user with an interface to the module that only provides the information needed and nothing more. This technique of information

hiding isolates the details of the mapping process inside of the interfacing module when there is structure clash.

Abstract Interfaces :

Parnas also describes the use of abstract interfaces [Parnas 77]. By abstract, Parnas is referring to the representation of several actual objects without specifics to any particular object. The essence of the real world is captured in the abstraction which can then be reused to other objects that the abstraction represents. The interface is the format of the information exchange between two software components along with all the assumptions that each makes about the other. An abstract interface is defined as a set of assumptions that represent more than one possible interface. An abstract interface models those properties that the interfaces hold in common and hides the differences.

Layered Interface Architecture

Two Layer Interfaces :

One way to map the mismatch between multiple paradigms is to build a two layer abstract interface (fig. 1). The upper level interface provides the actual interface to the user, it reflects the conceptual model appropriate to the application program, thus the application sees an appropriate model reflected in its interface. The interface uses the concepts of data abstraction to generalize the information crossing the interface, abstracting from particular characteristics of the external subsystem. The lower level interface reflects the semantics of the external system to the model transformation in the two layer interface, hiding the details of the external subsystem.

In addition to the semantic transformation provided by the abstract interface, there is a syntactic (language) transformation which must occur if the languages, in which the application and external subsystems are implemented, are different. This syntactic transformation may occur anywhere within the abstract interface. The syntactic transformation is often called the binding.

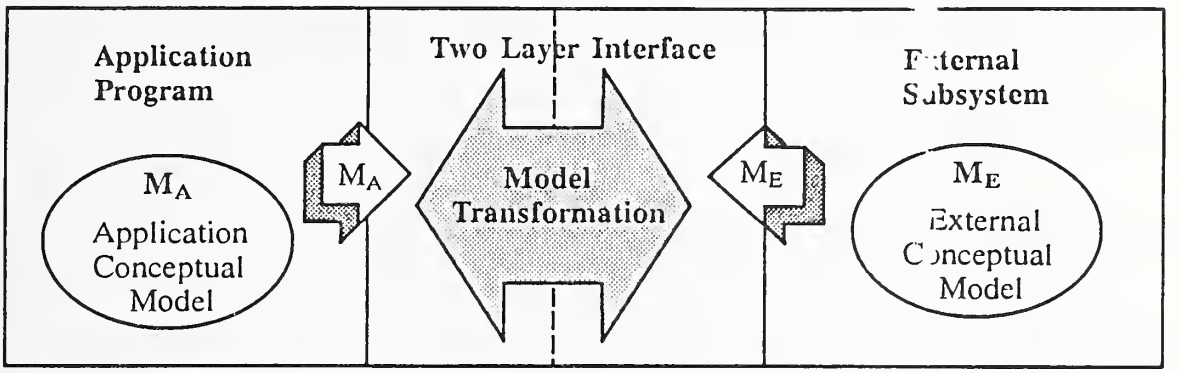


Figure 1

For example, in UNIX™, when a user requests a file be copied, the command cp along with the path to the file are all that the user has to supply. The operating system determines from the path which device the file resides on and makes the call to the appropriate device driver. The device driver is concerned with such things as: seek algorithms, inodes, motor speed, etc. The supplemental information and work involved in copying the file are transparent to the user. The upper level provides the file abstraction, the lower level the device read/write.

Another example is a two level architecture for communication within multi-Ada program systems [CASS 89]. The interprogram interface is defined in terms of remote procedure calls (RPC). In other words, to the application program the location of the called procedure is unknown. Again, the lower level is concerned with the details of the communications system, such as, sockets and communication protocols. The upper level provides an abstract interface to the user, taking care of the bundling of the message with logical source and destination names. The upper layer provides a procedural interface defining services used by the application, the lower layer a message passing transport service.

A third example of applying this architecture is the Ada/SQL interface architecture [Richardson, Wheeler and Ferrer 89]. The upper layer provides the semantic transformations while the lower layer provides the syntactic transformations. The lower layer contains the binding of SQL to Ada. In general, it takes care of the implementation details associated with the chosen binding and hides them from the application program. The Ada procedure calls to the lower layer reflect the syntax and semantics of SQL. The upper layer is the visible part of the interface to the Ada application programmer. The data abstractions

reflect the information needs of the Ada application program. The specifieation of the interface abstract away the SQL funetions used in the lower level. The upper level of the interfaee can be viewed as a retriever and maintainer of a persistent abstraet data type.

A forth example is the interface between the real world and a control system. In a eontrol system certain observable eharacteristics of the external world are monitored by sensor subsystems which understand the meaning of those characteristics. In this example the lower layer of the interface receives the sensor data coming in from the external world while the upper layer interprets the information into a meaningful representation. An example of this interface is a stop light control system [Wheeler 86]. The information from the sensors is passed to the lower layer of the interface. It is then sent to the upper layer whieh converts the information to a representation of traffic presence or absence in the various lanes. The upper level is coneerned with eontrol of the interseetion, the lower level with the sensors and relays.

As ean be seen in eaeh of these examples, the upper layer presents a model whieh is meaningful to the applieation, while the lower layer uses a model that is meaningful to the external subsystem.

Conclusion

This paper has presented a deseription of a two layered interfaee arehiteeture based upon Parnas' idea of abstraet interfaees. The architeeture allows two paradigms to eommunieate through a clean interface that presents to each side the desirable features of the other in the syntax and semantics of the user, with the upper layer presenting a model of the interfaces subsystem whieh is meaningful to the application, and the lower layer useing the model that is meaningful to the external subsystem. This architecture provides a basis for constructing language interfaees to object databases which addresses semantic, as well as syntactic, issuse in a coherent way.

References

[CASS 89] Common ATCCS Support Software Working Group–Architecture Subcommittee, *Requirements for Inter–Task Communication* (Draft), 1989.

[Hailpern 86] Hailpern, B., "Multiparadigm Languages and Environments", *IEEE Software*, January 1986.

[Jackson 75] Jackson, M., *Principles of Program Design*, Academic Press, New York, 1976.

[Parnas 72] Parnas, D. L., "On the Criteria to Be Used in Decomposing Systems into Modules", *Communications of the ACM*, December 1972.

[Parnas 77] Parnas, D. L., "Use of Abstract Interfaces in the Development of Software for Embedded Computer Systems", *NRL Report*, Number 8007, June 1977.

[Richardson, Wheeler and Ferrer 89] Richardson, J.D., T. J. Wheeler and A. Ferrer, "Ada and SQL: The Two Layer Approach", *Ada Technology Conference*, Atlanta, GA, March 1989.

[Wheeler 86] Wheeler, T. J., "An Example of the Developer's Documentation for an Embedded Computer System Written in Ada", *Ada Letters*, November–December 1986 and January–February 1987.

[Wheeler 87] Wheeler, T. J., *Memorandum for Record*, CECOM Software Technology Division, Information Processing Directorate, November 1987.

[Zave 89] Zave, P., "A Compositional Approach to Multiparadigm Programming", *IEEE Software*, September 1989.

[Zdonik and Maier 89] Zdonik, S. B. and D. Maier, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

Supporting User Views

Jonathan P. Gilbert

McDonnell Douglas Space Systems Company†
5301 Bolsa Avenue, MS. A95/J841/15-1
Huntington Beach, CA 92647

Abstract

Any emerging standard for object-oriented database systems must include a rich view support mechanism. A user view is a customized window into an application domain. It may be thought of as a simplifying abstraction which hides information that is not accessible to, needed or wanted by a particular user. Because they limit the information available from a given perspective, most views allow read-only access to a database. In this paper it is asserted that by generalizing object identity to include attributes and views – many view updates are made possible. An extended object structure and several categories of view transformation are also presented which allow all user views to be modeled within a single polymorphic database schema.

Introduction

The support of multiple, perhaps conflicting, user views is important in any multi user environment. Any standard for an object-oriented data model should include a mechanism view support. In this paper I will suggest an approach (based on [GILBERT90]) which could be incorporated into any object-based model.

The Basics

I will make some assumptions about the data model, all of which conform to the basic philosophy of object-orientation. There are two basic object types: classes and instances. There are also two basic types of relationship among objects called IS-A and attribute relationship. Class objects represent both a generic object (or *type* description) and a *set* of similar entities. They are arranged in one or more IS-A lattice such that those objects (nodes) near the top of the hierarchy contain a more general type description than nodes lower in the lattice. Instances are said to *belong to* a class if there is a path containing only IS-A arcs between the instance and the class; each instance is *owned by* exactly one class, meaning that the two objects are directly connected by a single IS-A arc.

---

†This is based on work performed while the author was a graduate student at the University of California at Irvine.

The Organization of a Database

Some useful definitions:

- A *schema* is a collection of classes and the attribute associations among them.
- *Data* is the collection of *instance objects* which are associated with the schema.
- A *view* is a schema and its associated data.
- A *database* is the combination of all of the views.

A database usually includes several views. An important relationship between a database and its views can be characterized by the following observation: the set of classes in any view is a subset of the set of classes contained in the database. Within a given database environment, lists of current views and classes are kept in a *global symbol table*.

Extensions to the Basic Paradigm

In the standard object-oriented model, each object has a private memory and a single protocol (a set of methods) which allow it to respond to requests from other objects (and users). Classes have exactly one name which must be unique in the environment. Similarly, attributes and methods must have a single shared name within a class definition. In order to support user views, several extensions will be made to the paradigm. Object and system structures must be enhanced in the following ways: (1) a layer will be added to the internal structure of objects, (2) a two-tiered global symbol table will be presented and (3) several categories of view transformation will be suggested. This will permit both classes and attributes to have one *or more* names and allow objects to support multiple "semi-public" protocols. These enhancements allow radically different views to be supported by a single *polymorphic* structure.

The Object Structure

This discussion will focus on class objects which have three major components: an *object description* (which contains the hidden internal representation of the object), a *view description* (which contains the semi-public interfaces) and *generic methods* (for retrieving and updating data). Figure 1 shows a pictorial representation of both the "standard" and extended object structures.

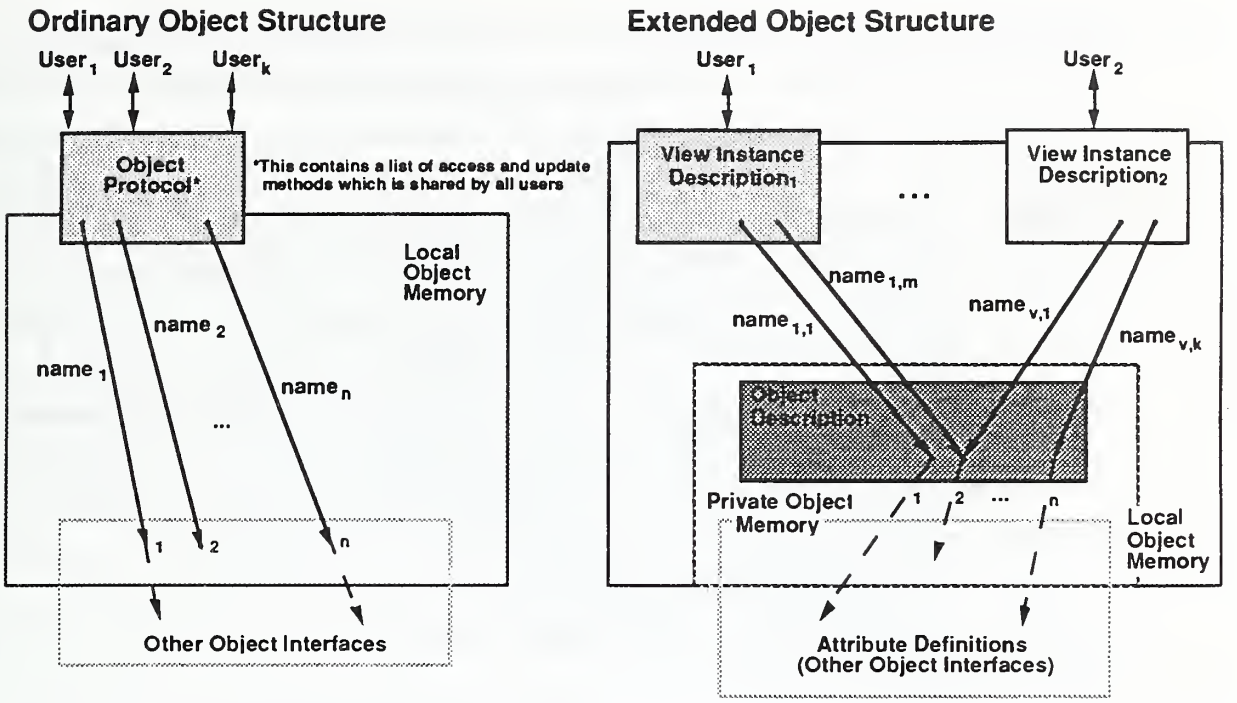


Figure 1: Object Structures

The Object Description

Object descriptions are automatically maintained by the system and are hidden from database users. An object description contains an object's identity and lists of attribute relationships, IS-A relationships and methods. Each property (method, IS-A and attribute relationship) also has its own unique identity. The same identity is associated with a property in *all* database objects which have inherited it from a common ancestor. There are two kinds of attribute: local attributes (which are entirely encapsulated in the object's local memory) and complex attributes which are either a calculated value (retrieved by executing a predefined method) or an object-oriented pointer (to an independent object). Within the object description attributes are also categorized by the properties associated with them. They may be mandatory (*key*) **or** not mandatory (*non-key*), *single* **or** *multi-valued*, and *changeable* **or** *not-changeable*. Finally, attributes have inverses which implies that they are bidirectional arcs between related objects in the schema.

The View Description

The view description contains one or more *view instance descriptions*. Each view instance description is a semi-public protocol for the object. Every view instance description has a unique system defined identity that is part of all the view's class

objects. Both the unique identities and the class' (printable) names are also found in the global symbol table. In addition, a view instance description contains lists of names and pointers to attribute and IS-A arcs and to methods which are contained in the *object description* and are *visible* through the view. It may also contain pointers to some of the attributes which are *not* visible from the current point of view but to which the *system* must have controlled access. All *invisible* (and some visible) attributes have a default object associated with them. Defaults are used to fill a role when a new object is created or for updating the attribute when the object is changed. For visible attributes defaults may be explicitly overridden by the user. Notice that this means that an object is updatable if all *key* attributes are included in the combined list of visible and invisible attributes. The view description may also include a natural language description of the view object, its structures and its role in the database enterprise. For example if, from a given point of view, an engine's primary function is to fill the propulsion system role of an automobile's description then this would be noted in a (written) description.

The Generic Methods

When an object receives a message, its first action must be to determine the requester's point of view. Once the view has been identified the request is processed *through* the appropriate view instance description. Restrictions on local attribute values are checked directly while restrictions on complex attributes cause subqueries to be spawned and *independently* processed. Intermediate results are collected and the request is either propagated to descendents (in the IS-A hierarchy) or the result is returned to the source of the query. Note, this is a very high level sketch of an *asynchronous message-driven* data retrieval and update mechanism. This mechanism, which requires *no centralized control* is presented in detail in [GILBERT90].

The Global Symbol Table

A two-tiered global symbol table structure is maintained. There is a single *global symbol table* and one or more *view model object tables*. Although there are usually several view model object tables, there is exactly one for each view. Because users may share a view, there may be several entries in the global symbol table associated with a single view model object table.

Figure 2 shows a global symbol table and its associated view model object tables. This particular table contains three views. The first column of the global symbol table contains identifiers, each of which is associated with a particular view of the database.

The second and third columns contain (internal) system maintained view identities and pointers to view model object tables respectively.

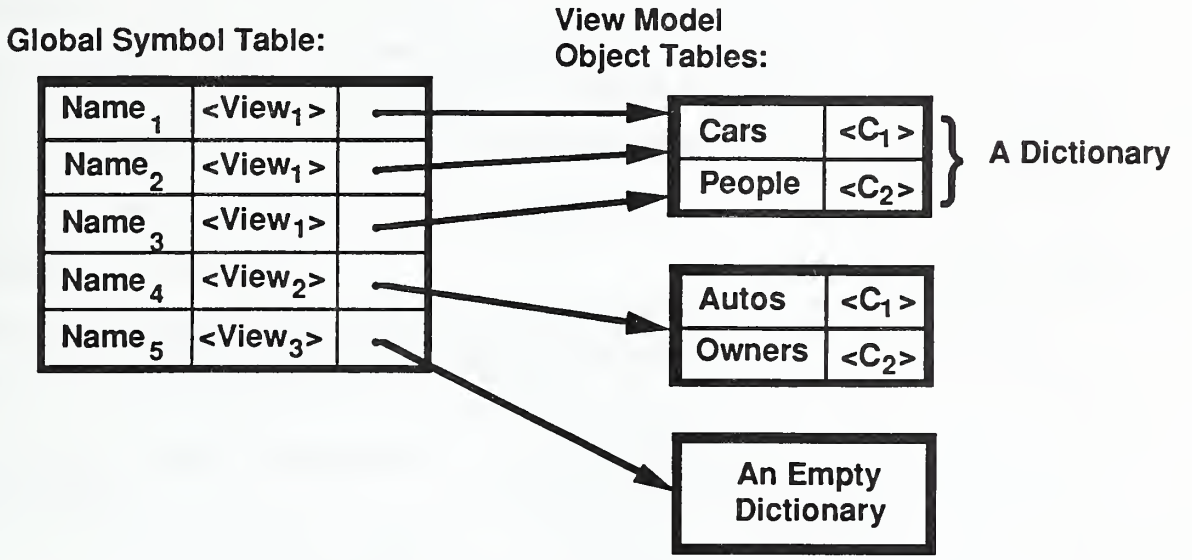


Figure 2: The Global Symbol Table

Each view model object is associated with a single view. Their first columns contains view sensitive external names which are associated with the internal name of classes in the database schema. In figure 2, <$View_1$> and <$View_2$> are fully instantiated user views which share the same classes whose identities are <C_1> and <C_2>. Their global symbol table entries point to particular view model objects. <$View_3$>, on the other hand, is in the process of being created – its symbol table entry currently refers to a default view model object table (an empty *dictionary*[1]).

View Transformations

View transformations are the mechanisms by which user views are created and customized. They are used to create global symbol table entries and view instance descriptions for new user views. In this forum there is only enough space to discuss the various categories of view transformation. Significantly more detail can be found in [GILBERT90]. The basic view forming transformations fall into three major categories: (1) graph tailoring methods, (2) operations on the IS-A hierarchy and (3) procedures for customizing attribute relationships. Graph tailoring transformations treat a view as a single object; they create, record or delete an entire view schema. Once a (virt) view schema has been created, the other transformations can be applied to custo

[1]This concept is borrowed from Smalltalk [GOLDBERG83]. A dictionary is a table which allows *associative* (key word) lookup.

the schema so that it meets the particular needs of the user. These transformations affect individual classes and the IS-A of attribute hierarchies *within* a particular view schema. Objects may be cloned or renamed and attached, removed or hidden within the IS-A hierarchy. Attributes can be renamed, removed, restricted, inserted or moved. All the transformations are very simple and the changes caused by applying them are "localized" – i.e. they change a particular user's view but have absolutely no effect on any other database user.

Concluding Remarks

A standard for user view support will be a necessary part of any object-oriented database standard. The mechanisms described in this paper address one aspect of this problem. Another problem, which is closely related to view support but beyond the scope of this paper, is the identification of semantic groupings and derived classes. This topic has been discussed in [GILBERT88, GILBERT90, HEILER90]. Several types of derived data have already been identified including *collections* (derived from the union of base classes – automatically maintained by the system based on a predefined rule), *categories* (a user controlled collection – there is no predefined membership rule), *power sets* (each instance is a category or a collection) and *joins*. The object database standard should include a set of semantic groupings in addition to the view abstractions presented here.

References

[GILBERT88] GILBERT, J. P. and BIC, L., Asynchronous Data Retrieval from an Object-Oriented Database. In *Proceedings of the European Conference on Object Oriented Programming, Oslo, Norway*, Springer Verlag, 1988.

[GILBERT90] GILBERT, J. P., PolyView: An Object-Oriented Data Model For Supporting Multiple User Views. PhD dissertation available as Technical Report #90-05, Department of Information and Computer Science, University of California, Irvine, 1990.

[GOLDBERG83] GOLDBERG, A. AND ROBSON, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass., 1983.

[HEILER90] HEILER, S. AND ZDONIK, S., Object Views: Extending the Vision. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, CA., 1990.

NIST-114A
(REV. 3-90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4488

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
January 1991

1. TITLE AND SUBTITLE

Proceedings of the Object-Oriented Database Task Group Workshop
Tuesday, October 23, 1990; Chateau Laurier Hotel; Ottawa, Canada

2. AUTHOR(S)

Elizabeth N. Fong, Editor

3. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
GAITHERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED

4. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

National Institute of Standards and Technology
Gaithersburg, MD 20899

0. SUPPLEMENTARY NOTES

1. ABSTRACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

This report constitutes the proceedings of a one-day workshop on standardization of object database systems held at the Chateau Laurier Hotel, Ottawa, Canada, on October 23, 1990. The workshop was sponsored by the Object-Oriented Database Task Group (OODBTG) of the ANC/X3/SPARC Database Systems Study Group (DBSSG).

This workshop, held on the third day of the Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA), was the second attempt to solicit public input to identify what aspects of object database systems may be candidates for consensus that can lead to standards. The first companion workshop was held on May 22, 1990, in Atlantic City, New Jersey, coincident with the International Conference on Management of Data (SIGMOD).

The workshop goals focused on concrete proposals for languages or module interfaces, exchange mechanisms, abstract specifications, common libraries, or benchmarks. The workshop announcement also solicited papers on relationships of object database systems capabilities to existing standards, including assertions that question the wisdom of standardization.

This proceedings consist of 13 position papers covering various aspects where standardization on object database systems may be possible.

KEY WORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

database; database management system; DBMS; data model; object-oriented; OODB; programming languages; standards.

AVAILABILITY

[X] UNLIMITED

[ ] FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

[ ] ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

[X] ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
158

15. PRICE
A08

ECTRONIC FORM