Distributed and Parallel Databases 2 (1994), 101–126

Schema Evolution and Integration

STEWART M. CLAMEN clamen@cs.cmu.edu
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213

Abstract. Providing support for *schema evolution* allows existing databases to be adjusted for varying roles over time. This paper reflects on existing evolution support schemes and introduces a more general and functional mechanism to support schema evolution and *instance adaptation* for centralized and distributed object-oriented database systems. Our evolution support scheme is distinguished from previous mechanisms in that it is primarily concerned with preserving existing database objects and maintaining compatibility for old applications, while permitting a wider range of evolution operations. It achieves this by supporting schema versioning, allowing multiple representations of instances to persist simultaneously, and providing for programmer specification of how to adapt existing instances. The mechanism is general enough to provide much of the support necessarily for *heterogeneous schema integration*, as well as incorporating much of the features of object migration and replication.

Keywords: schema evolution, heterogeneous schema integration, object-oriented database systems, instance adaptation

1. Introduction

Object-oriented database systems (OODBS) are considered *distributed* when they are composed of multiple database servers at multiple sites, connected via a network. Distribution is motivated by both economics and pragmatics. Economically, it allows for a sharing of resources: a collection of networked computers sharing the work and the storage load of a large and active database. Pragmatically, distribution can improve reliability and accessibility to the database by providing continued access to much of the database in spite of isolated hardware failures. Replication of the data can further improve database availability by reducing the likelihood that a particular piece of data would only exist at inaccessible sites.

Distributed object-oriented database systems (DOODBS) are (minimally) composed of:

- A collection of network-accessible computing sites,
- A **schema**: a set of class definitions, including an inheritance hierarchy. A class definition defines a set of typed attributes (the representation) and a set of *methods* (the interface).
- A **database**: a set of persistent objects, residing on any of the distributed sites, each an instance of a class defined in the schema, and each with a unique identity [19].

- A set of application programs, interacting with the objects in the database via the interfaces defined in the schema.

DOODBS can be categorized by the degree of autonomy exhibited or permitted each constituent database. They can also be classified with respect to the heterogeneity of their data models and their choices with respect to a schema [27].

This paper distinguishes between DOODBS with a **homogeneous** schema and those with **heterogeneous** schemas.[1] Homogeneous DOODBS, although implemented on multiple database servers, share a common data representation and interface. Each database making up a heterogeneous system has its own schema. For simplicity, we will assume that the only source of heterogeneity is schematic; we will not address the potential differences in data or query models that multidatabase systems might possess.

The primary challenge in the design and implementation of a distributed database system is in making its use (by the user or application program) as transparent as possible. Ideally, a user should be able to interact with the set of distributed databases as if they constituted a single, local one. For **heterogeneous** DOODBS, this means that some mechanism must exist to relate the various schemas and their instances. This process, called **schema integration**, has been addressed in the context of distributed relational systems (RDBS). The information that relates the heterogeneous schemas to each other can be termed the **integration schema**. A sample set of heterogeneous schemas are presented in Figure 1.

1.1. Schema evolution and distributed object-oriented databases

Database systems exist to support the long-term persistence of data. It is natural to expect that, over time, needs will change and that those changes will necessitate a modification to the interface for the persistent data. In an object-oriented database system, such a situation would motivate an evolution of the database schema. For this reason, support for **schema evolution** is a required facility in any serious OODBS.

OODBS were motivated by research into the development of applications centered around design tasks. The *design applications*, exemplified by CAD/CAM systems, multimedia and office automation facilities, and software engineering systems, are characterized by their combined need for database and programming language functionality. But these design applications, which use the persistent store as a medium for the sharing of complex information, are all the more susceptible to schema changes, as the design process is an evolutionary one [16].

Many of the tasks in this domain can benefit from the distribution of the persistent data repository. With the large quantities of complex information that might be involved, distribution promotes the sharing of the expense of maintaining

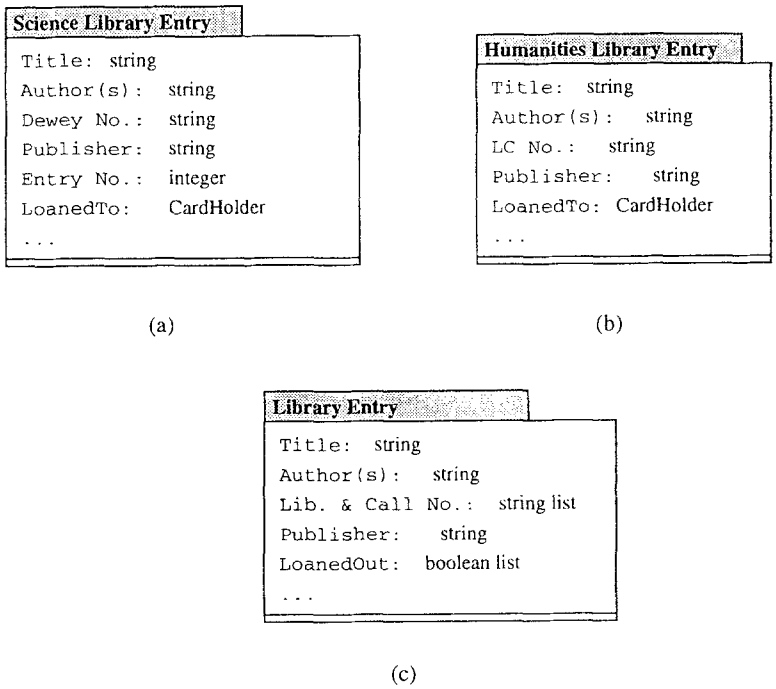


Figure 1. Two schemas to integrate: Here are two (simplified) schemas representing the local card catalogues of two distinct university libraries. Integrating the two base schemas results in a unified card catalogue for the campus, for use by the students. The result would see one `Library Entry` instance for each `Science Library Entry` or `Humanities Library Entry` instance.

the hardware resources. While communication among users via the database is a necessity, much of the work in design applications is localized, composed of long-lived, personal (design) transactions. Reliability (in the presence of potential communication failure) and efficiency could be gained by maintaining a proximate relationship between the data and most likely user. Also, there are management benefits to dividing the facility into smaller components.

Our evolution support scheme is distinguished from previous mechanisms in that it is primarily concerned with preserving existing database objects and maintaining compatibility for old applications, while permitting a wider range of evolution operations; previous schemes tend to support a limited variety of evolutions and rarely provide application compatibility support. Compatibility is supported by versioning the schema and allowing multiple representations of objects to persist simultaneously. The range of supported evolutions is increased by allowing the programmer to specify the relationship between the old and new object representations.

1.2. Guide to the paper

This paper is structured as follows: Section 2 discusses the issues associated with schema evolution and presents some previous approaches to the problem. Section 3 presents our new, flexible evolution scheme and explains how it is general enough to support much of what is required to support heterogeneous schema integration as well. Section 4 relates the scheme to some other DOODBS issues such as *object migration* and *replication*. Section 5 provides concluding arguments and outstanding research issues.

2. Schema evolution in object-oriented databases

When the schema changes so does the application/database interface, possible leaving incompatible elements on both sides of the abstraction barrier. We will focus on the problem of managing those (pre-)existing database objects, what we call the **instance adaptation problem**. In this section, we will examine the limitations of existing (schema) evolution and (instance) adaptation schemes. Towards the end of the section, we will illustrate how the schema evolution problem is very similar to the problem of heterogeneous database schema integration.

2.1. Existing approaches

Two general instance adaptation strategies have been identified and implemented by various OODB systems. The first strategy, **conversion**, restructures the affected instances to conform to the representation of their modified classes. Conversion is supported by the ORION [2, 20] and GemStone [8] systems.

The primary shortcoming of the conversion approach is its lack of support for **program compatibility**. By discarding the former schema, application programs that formerly interacted with the database through the changed parts of the interface are now obsolete. This is an especially significant problem when modification (or even recompilation) of the application program is impossible (e.g., commercially distributed software).

Rather than redefining the schema and converting the instances, the second strategy, **emulation**, is based on a **class versioning scheme**. Each class evolution defines a new version of the class and old class definitions persist indefinitely. Instances and applications are associated with a particular version of a class and the runtime system is responsible for simulating the semantics of the new interface on top of instances of the old, or vice versa. Since the former schema is not discarded but retained as an alternate interface, the emulation scheme provides program compatibility. Such a facility has been developed for the Encore system [29].

Encore pays for this additional functionality with a loss in runtime efficiency.

Under a conversion scheme, the cost of the evolution is a function of the number of affected instances. Once converted, an old instance can be referenced at the same cost as a newly created one. However, the cost of emulation is paid whenever there is a version conflict between the application and a referenced instance.

We feel however that program compatibility among schema versions is a very desirable feature under certain circumstances. It can be of great utility in situations where the database is shared by a variety of applications, as in computer-aided design or office automation systems, when the database acts as a common repository for information, accessed by a variety of applications. As these types of applications are also those which benefit from distribution, we see that compatibility support in DOODBS is all the more desirable.

Our scheme supports program compatibility by maintaining multiple versions of the database scheme. Old programs can continue to interact with the database (on both new instances and old) using the former interface. Rather than emulating the evolved semantics all at runtime, efficiency is gained by representing each object as an instance of each version of its class. In this manner, our system effects a compromise between the functionality of emulation and the efficiency of conversion.

Another failing common to the conversion-based evolution facilities is the limitations placed on the variety of schema evolutions that can be performed. Most existing systems restrict admissible evolutions to a predefined list of schema change operations (e.g., adding/deleting an attribute or method from a class, altering a class's inheritance list). The length of this list might vary from system to system, but they are all similar in the way they support change: *The set of changes that can be performed are those which require either a fixed conversion of existing instances or no instance conversion at all.* Unfortunately, change is inherently unpredictable. A desired evolution is sometimes *revolutionary* and under such circumstances, these systems prevent the database programmer from performing the desired changes.

We are interested in supporting evolution in a liberal rather than a conservative fashion; rather than the system offering a list of possible evolutions to the programmer, the programmer should be able to specify arbitrary evolutions and rely on the system for assistance and verification. Change is a natural occurrence in any engineering task, and engineering-support systems should help rather than hinder when an evolution is required.

Although Encore's emulation facility restricts the breadth of class evolution that can be installed, the restrictions are of a different form. Since instances, once created, cannot change their class-version, evolutions that require additional storage for each instance cannot be defined.

In the next section, we present a model for specifying schema evolutions and instance adaptation strategies. Our system supports program compatibility, accepts a larger variety of evolutions than existing systems, and supports a variety of options to make it more efficient than the pure emulation facility of Encore.

A number of evolution support systems have been incorporated into existing systems or proposed in the literature. Notable representatives are described below.

2.1.1. ORION. The most ambitious and effective example of a schema evolution support facility is that provided by the distributed (homogenous) OODB system ORION [2, 20, 21]. ORION provides a taxonomy of schema evolution operations (e.g., add a new class; add a new class attribute; rename a class attribute; change the implementation of a class method.) It also defines a database model in the form of invariants that must be preserved across any valid evolution operation and a set of rules that instruct the system how best to maintain those invariants. Under this model, a schema designer specifies an evolution in terms of the taxonomy and the system verifies the evolution by determining if it is consistent with the invariants and then adjusts the schema and database according to the appropriate rules.

ORION can only perform those evolutions for which it has a rule defined. The set of rules is fixed. For example, changes to the domain of an attribute of a class are restricted to generalizations of that domain. This restriction exists because there is no facility in ORION's evolution language for explaining how to "truncate" attribute values that are now outside the attribute's domain. (Generalizations of the attribute domain are allowed since this evolution does not require existing instances to be modified.)

In ORION, evolutions are performed on a unique schema. Instances are converted lazily. There is no compatibility support for old programs and, depending on the evolution, information contained in the instances might be lost at conversion time. (e.g., deletion of an attribute.)

The last implementation of ORION, ORION-2[22], supported personal databases in association with a central public database. Personal subschema could be devloped but could not be defined in opposition to the information contained in the central (public) schema. When information is moved from a personal to the central database, the personal subschema is merged into the central schema.

2.1.2. Encore. Encore implements emulation via user-defined exception handling routines. Whenever there is a version conflict between the program and the referenced instance, the routine associated with that method or instance (and those pair of versions) is called. The routine is expected to make the method's invocation conform to the expectations of the instance or make the return value from the method invocation consistent with the expectations of the calling program, whichever is appropriate. It is known, however, that certain evolutions cannot be modeled adequately under this scheme. The problem stems from the fact that each object can only instantiate a single version. If an evolution includes the addition (subtraction) of information (e.g., the addition (deletion) of an attribute), there is no place for older (newer) instances to store an associated value. The best a programmer could do in such a system is associate a default

attribute value for all instances of older (newer) type-versions by installing an exception handling routine to return the value when an application attempts to reference that attribute from an old (new) instance [29].

2.1.3. The common lisp object system. CLOS [17, 30], while not an OODB system, provides extended support for class evolution nonetheless. As Common Lisp system development is performed in an interactive context, class redefinition is a frequent occurrence. Rather than discard all existing instances, CLOS converts them according to a policy under the control of the user. The default policy is to reinitialize attribute values that no longer correspond to the attribute domain, and to delete attribute slots that are no longer represented in the class definition. Users can override this policy by defining their own method that is called automatically by the system. This method is passed as arguments the old and new slot values, so relationships between deleted and added attributes can be enforced [30, p.859].

2.1.4. Other approaches. Bertino [3] presents a schema evolution language which is an OODB adaptation of the view mechanism found in many relational database systems. Her primary innovations are the support of inheritance and *object IDs* (OIDs) for view instances, two important characteristics of OODB models that are not present in the relational model. View instances with OIDs are physically realized in the database, enabling the view mechanism to support evolutions that specify the addition of an attribute, as envisioned by Zdonik [34]. However, Bertino's scheme focuses on how evolutions affect the schema. It is not concerned explicitly with the effects upon the instances nor with compatibility issues.

Zicari proposed [35, 36] a sophisticated evolution facility, providing an advisory program to determine at evolution time whether the evolution is consistent with interclass and method dependencies. Evolution transactions are introduced to allow for compound evolution operations. However, Zicari's lack of concern for instance adaptation is evident; by defining the attribute-renaming evolution as the atomic composition of the attribute-delete and attribute-add operations, his scheme fails at the instance level.

Monk's CLOSQL [25] implements an class versioning scheme, but employs a conversion adaptation strategy. Instances are converted when there is a version conflict, but unlike ORION, CLOSQL can convert instances to older versions of the class if necessary.

Lerner's OTGen design [23] addresses the problem of complex evolutions requiring major structural conversions of the database (e.g., information moving between classes, sharing of data using pointers) using a special-purpose language to specify instance conversion procedures. As it was developed in an integrated database context, where the entire application set is recompiled whenever the schema changes, versioning and compatibility were not considered. However, Lerner's language supports a variety of evolutions and associated adaptations

that are not addressed in many other papers, most notably evolutions that alter the structure of shared component objects.

Bratsberg [6, 7] has been developing a unified semantic model of evolution for object-oriented systems. Similar to our work, compatibility for old clients is described in the context of relations, maintaining consistency between views.

One significant difference between our respective threads of research is our concentration on the variety of adaptation strategies and representations for the (possibly) multifaceted instances. This is reflected in this paper's discussion of the range of possible adaptation strategies, depending on the (expected) access patterns of the affected instances.

2.2. Schema modification versus class versioning

The schema evolution support provided by such systems as ORION and GemStone is restricted to what Kim calls **schema modification**, that is, *the direct modification of a single logical schema* [21] When only one database schema exists, it is appropriate for the system to convert all existing instances. From a database consistency perspective, it must *appear* that all instances have been converted when the evolution operation is applied.[2] In fact, we would claim that it is the *only* sensible approach.

As has already been stated, however, conversion might render the instances inaccessible to applications that had previously referenced them. The adaptation strategy converts the instances but does not alter procedural references. Thus, application programs written and compiled under the old schema may now be obsolete, unable to access either the old, now converted, instances, or the ones created under the new schema.

A reasonable direction of research here would be to provide some automated mechanisms to assist with program conversion; it is an active line of research [1, 13]. In the OODB context, some work has been conducted at providing support to alert the programmer about the procedural dependencies of their evolution operation [10, 33]. But this is not the only possible solution. Rather than adjust programs to conform to the data, it would seem easier to adjust the data to conform to the existing programs. Also, it is not always possible to alter, or even recompile, programs (e.g., commercially available software). This lack of compatibility support is our primary motivation for adopting a class versioning design for evolution management and support (Section 3).

Under a class versioning scheme, multiple interfaces to a class, one per version, are defined. When compiled, application programs are associated with a single version of each of the classes it refers to; a *schema configuration*, if you will. With the database populated with instances of multiple versions of a class, the runtime system must resolve discrepancies between the version expected by the application and that of the referenced instance.

It is worth observing here that schema versioning introduces a notion of

schema heterogeneity in the absense of distributed, autonomous databases. This characteristic will be elaborated upon later (Section 2.4).

2.3. Schema evolution in distributed object-oriented databases

Distribution of a database creates new implementation issues with respect to schema evolution support, and increases the importance of others.

Any OODBS requires the persistent management of the database schema(s). In a distributed environment, the common schema (the only schema, in the case of homogeneous systems, and the integration schema for heterogeneous systems) must remain as available as possible, so maintaining a copy with each database server is a reasonable decision. Changes to this schema would require updates to be propagated to every server, although these changes could be installed lazily, thereby obviating the need for all servers to be accessible at evolution time.[3]

The distribution and improved ease of remote access to the database strengthens the motivation for backward compatibility support. The larger the community sharing the system and schema, the more frequent and less integrated the changes, the greater the need to keep evolution dependencies (both applications and existing persistent objects) to a minimum.

2.3.1. Heterogeneous schema evolution. When the distributed collection of databases represent different schemas, the means and affects of schema evolution are altered. The primary difference is the existence of a schema hierarchy. Evolutions to the integration schema are distinguished from evolutions to local schemas.

When a local schema is modified, a change to the integration schema might become necessary. But since the role of the integration schema is to present a common interface for distributed applications, only the implementation of the integration schema, and not its exported interface, would need any modification. As each database is considered autonomous, a local evolution should not affect the objects in remote databases.

Evolution of the integration schema could be performed independently of the various distributed schemas. However, such evolutions from above might require coincident (or previous) evolutions on the associated databases. Such cross-administrative evolutions require extensive coordination, much like evolutions in the absence of an intrinsic evolution facility.

2.4. Heterogeneous schema integration

Let us review the purpose of schema integration in the context of heterogeneous DOODBS. A system is composed of a number of distinct databases, each with its own schema and its own objects. The *integration schema* presents a single schema to applications for accessing these diverse databases. (Figure 2)

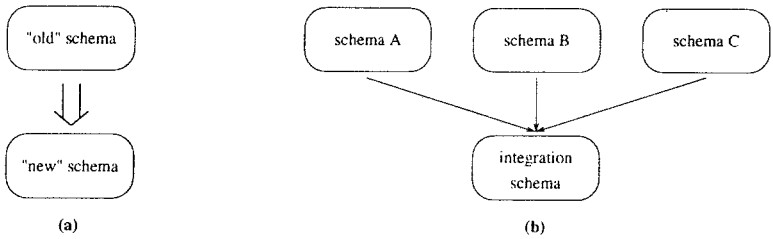


Figure 2. Evolution and integration: Both provide a mechanism for relating schemas: (a) evolution involves a migration from one schema to a new, unpopulated one; (b) integration coalesces a set of schemas into one, and is complicated by the fact that instances in the various source schemas may need to be merged (virtually or actually) into a single one in the target schema.

Schema versioning technology could be beneficial in this context.

Consider the heterogeneous database problem in the absence of distribution: imagine all the databases collected into a single database, with one large schema that is the disjoint union of the distributed schemas (so naming conflicts are avoided) and the integration schema. (Note that the classes that make up the integration schema are virtual, and lack instances of their own.) If our OODBS supported schema versioning, and supplied class emulation facilities similar to that provided by Encore, we could implement integration (i.e., unified access across the distributed schemas) by writing routines to emulate the integration schema in terms of each of the formerly distributed schema! (See Table 1 for example.)

Integration and evolution are actually two specializations of the same problem: that of relating different schemas that model parts of the same domain. Their basic distinguishing feature is the *currency* of the various schemas. Integration is the "merging" (either via conversion or emulation) of a set of existing, equally current schema and associated objects. Evolution is motivated by the desire to move from one schema (and database) to a "new and improved" one. Note, however, that evolution need not always be motivated by "progress." We can easily contemplate the "devolutions" motivated by backward (application) compatibility (e.g., CLOSQL, p.8), or "backing out" of an ambitious, yet ill-conceived, upgrade.

It stands to reason then, that a general mechanism could be developed to assist with both these tasks [7]. Such a scheme (first presented in the context of evolution) appears in the following section.

3. Supporting conversion and compatibility

Section 2.2 described the advantages of a schema versioning approach to evolution. Herein, we sketch an implementation for such a scheme.

Table 1. Emulating Integration: a rough sketch of how to implement the integration schema from Figure 1 using Encore-like emulation routine. The table illustrates how to emulate the attribute read calls for the universal `Library Entry` in terms of the distributed `Science Library Entry` and `Humanities Library Entry` schemas. (Collecting multiple book entries is omitted for simplicity.)

To emulate Library Entry ...	from Science Library Entry	from Humanities Library Entry
Title	Title	Title
Author	Author	Author
Lib & Call No.	"SCI" + Dewey No.	"HUM" + LC No.
Publisher	Publisher	Publisher
LoanedOut	LoanedTo $\neq$ NIL	LoanedTo $\neq$ NIL

3.1. Database model

As a basis for our discussion, we will employ a simplified object-oriented database model.

All objects found in the database are instances of classes. A class is record type of attributes and methods. An attribute is a private, named, typed representation of state, which can be accessed only by class methods. Methods are descriptions of behaviour and can be public or private to the class. Under these restrictions, the set of public methods describe the *interface* of the class, while the attributes model its *state*.

Classes are arranged in a type hierarchy (actually a directed, acyclic graph): a class's interface must be a generalization of the union of its superclasses (supertypes); the specification of each inherited method must be at least as general as those of its superclasses. (It should also be stated that methods must have semantics consistent with those of their class's supertypes.) Instances of a class that is a declared subtype of another class can be referenced as if they were instances of the superclass. We do not consider class inheritance in our model.

Unlike the inheritance mechanisms provided by many object-oriented languages and OODB systems, our model's subtyping mechanism does not compromise modularity but continues to provide some of the advantages of type hierarchy identified by Liskov [24]. The maintenance of class modularity in this regard greatly simplifies the evolution and adaptation model described in this paper.

In addition to the classes, other supported types include primitive types (e.g., integer, floating point number, character) and arrays.

The set of defined classes for each member database comprise the local database schema. Each class has an associated unique ID. All objects found in the database are instances of classes. Each instance is identified by a unique *Object ID* (OID), and is tagged with its *Class ID* (CID).

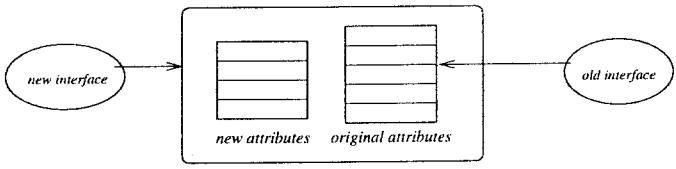


Figure 3. Zdonik's Wrapping Scheme: as in the Encore design, multiple interfaces to the class are preserved. Here, extra space is allocated for the attributes added as a result of the evolution, and applications can access the instance through either the old or new interfaces.

For integration purposes, we will assume that there exists a global, integration schema, and a global method for identifying specific objects. These features will be elaborated upon in the course of this paper.

3.2. Objects instantiating multiple class-versions

Under the original Encore schema evolution support design [29], instances never change their type-version. Aware of the restrictions this causes (see previous section), Zdonik proposed a scheme whereby an existing instance can be "wrapped" with extra storage and a new interface, enabling it to be a full-fledged instance of a new type-version [34]. While still accessible through its original interface/version, the wrapped object can also be manipulated through the new interface. Thus, if the class evolution specifies the addition of an attribute, the wrapping mechanism could allocate storage for the new slot in existing instances, without denying backward compatibility (Figure 3).

Our scheme is a generalization of this approach, and resembles the view abstraction mechanism proposed by JANUS [14], as well as the *type conformance* principle introduced as part of the Emerald data model [4]. Instead of supporting a single interface, we can provide multiple interfaces to instances. Much as each class has multiple versions, each instance is composed of multiple **facets.** Theoretically, each facet encapsulates the state of the instance for a different interface (i.e., version). The representation of these instances is, abstractly, a disjoint union of the representation of each of the versions, and it is useful to consider the representation as exactly that. As will be explained later, however, a wide variety of representations are possible.

As an example, consider a class `Undergraduate`, originally including attributes `Name`, `Program`, and `Class`, and a new version of the class with the attributes `Name`, `Id Number`, `Advisor`, and `Class Year`. (`Class` is one of {Freshman, Sophomore, Junior, Senior}, while `Class Year` is the year the student is expected to graduate.) `Degree Pgm` is the degree program in which the student is enrolled, and `Advisor` is his academic advisor. While instances of `Undergraduate` in the database will contain all seven distinct attribute slots, any particular application will be

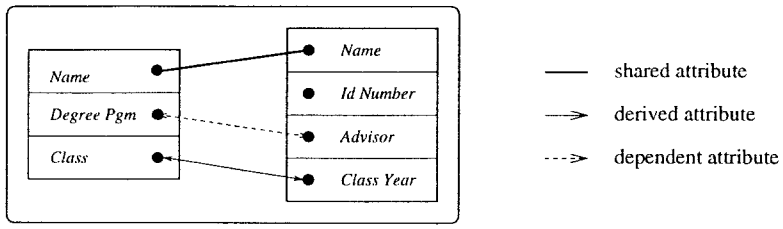


Figure 4. Disjoint union representation of the versioned class `Undergraduate`.

restricted to one version and thus only have explicit access to one facet.

In reasoning about the relationship between any two versions of a class,[4] it is useful to divide the attributes into these four groups:

Shared: when an attribute is common to both versions,[5]
Independent: when an attribute's value cannot be affected by any modifications to the attribute values in the other facet.
Derived: when an attribute's value can be derived directly from the values of the attributes in the other facet,
Dependent: when an attribute's value is affected by changes in the values of attributes in the other facet, but cannot be computed solely from those values.

In our example (Figure 4), the `Name` attribute is shared by the versions, while `Id Number` is independent. `Class` and `Class Year` are both derived attributes since, given the current date, it is possible to derive one from the other. `Advisor` is a dependent attribute, since a change in `Degree Pgm` might necessitate a change in advisor. Likewise, `Degree Pgm` is a dependent attribute, since a change in advisor might imply that the student has switched degree programs.

Zdonik et al. [29, 34] almost always cite evolutions involving independent or derived attributes in their examples. The original Encore emulation scheme is adequate for supporting evolutions that introduce shared and derived attributes. Zdonik's wrapping proposal addresses the problems associated with independent attributes. Our scheme, however, will provide a mechanism for managing class evolutions that include the former three categories plus dependent attributes.

3.3. Specifying an adaptation strategy (with example)

Given two versions of a schema (simplified here to a versioned class), we are required to categorize the attributes (of each class-version) accordingly, and associate adaptation information with each of them: for shared attributes, identifying its "synonym" in the other version; for derived attributes, a function

for determining the attribute value in terms of attribute values in the other facet; for dependent attributes, a function in terms of the attributes of *both* facets. Independent attributes require no additional information.

A relation for a version in terms of the other version can be generated given the supplied attribute-wise information. For backward compatibility to be supported, dependency relationships must exist in both directions between the two class-versions. In such cases, a correctness constraint exists, i.e., the version-wise relation from version A to version B must be the inverse of the relation from B to A. (Note that determining if the two relations are inverse of each other is analoguous to the halting problem in general.)

Representing the class instances as a disjoint union of the version facets, as described earlier, consistency between the facets can be maintained according to the following procedure:

> *Whenever an attribute value of a facet is modified, those attributes in the other facet that depend on it must be updated. For shared attributes, the new value is copied; for dependent and derived attributes, the dependency functions are applied and the result written into the (attribute) slot in the other facet.*

The remainder of this subsection consists of an example:

> Consider the `Undergraduate` class versions introduced earlier. The derivation function for `Class Year` is

$$\texttt{Class Year} = \begin{cases} \text{cy} + 3 & \text{if } \texttt{Class} = \texttt{Freshman} \\ \text{cy} + 2 & \text{if } \texttt{Class} = \texttt{Sophomore} \\ \text{cy} + 1 & \text{if } \texttt{Class} = \texttt{Junior} \\ \text{cy} & \text{if } \texttt{Class} = \texttt{Senior} \end{cases}$$

> Where cy is the current year.
> Likewise, the derivation for `Class` is

$$\texttt{Class} = \begin{cases} \texttt{Freshman} & \text{if } \texttt{Class Year} = \text{cy} + 3 \\ \texttt{Sophomore} & \text{if } \texttt{Class Year} = \text{cy} + 2 \\ \texttt{Junior} & \text{if } \texttt{Class Year} = \text{cy} + 1 \\ \texttt{Senior} & \text{if } \texttt{Class Year} = \text{cy} \end{cases}$$

> The `Advisor` attribute is dependent upon the value of the `Degree Pgm` attribute, but not completely derivable. A reasonable dependency function is:

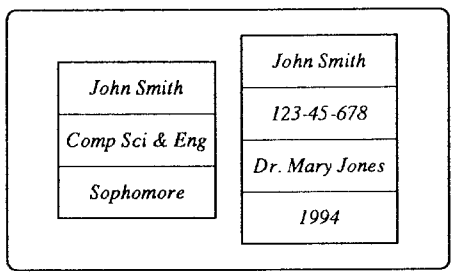
$$\texttt{Advisor} = \begin{cases} \texttt{Advisor} & \text{if } \texttt{Advisor} \in \text{Program faculty} \\ \text{nil} & \text{otherwise} \end{cases}$$

Similarly:

$$\texttt{Degree Pgm} = \begin{cases} \text{Program of } \texttt{Advisor's field} & \text{if singular} \\ \text{Existing value of } \texttt{Degree Pgm} & \text{if } \texttt{Advisor} \in \text{Program faculty} \\ \text{nil} & \text{otherwise} \end{cases}$$

The dependency functions for each adaptation "direction" satisfy the inverse-relation constraint introduced earlier.

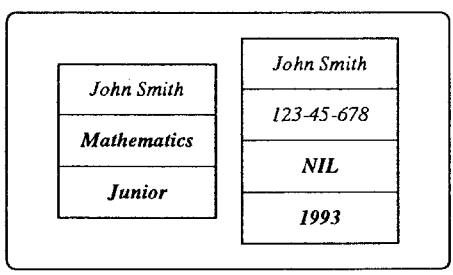
Consider a multifaceted instance of `Undergraduate`, represented graphically as follows:



Imagine that John Smith returns to university after his first summer vacation and wishes to change to the undergraduate math program. Also, he had taken some summer classes that have given him enough credits to graduate a year early. The change to his data record are recorded through an application program employing the first version of the `Undergraduate` class. The system must now propagate those modifications to the second facet, using the dependency functions from above.

Since there is not enough information to derive it, the student's advisor will have to be filled in later.

Applying these functions in concert with the desired changes to John Smith's record, the multifaceted instance becomes



3.4. Representing multifaceted instances

In the previous section we described the semantics of our schema versioning scheme. In this section we address some of the representation issues.

We begin with the simple and direct implementation: class evolutions are defined by creating a new version of the class; a new facet (corresponding to the new version) is associated with each instance of the class and initialized according to the programmer-defined adaptation specification.[6] Each application program interacts with the instances through a single version (interface) and modifications to attribute slots on the primary facet are immediately propagated to the other facets, using a mechanism similar to the trigger facility found in many relational and AI database systems [12, 31].

This simple implementation can be made more efficient. The most obvious target for improvement is how new facets are added. The allocation and initialization of new facets for existing instances at evolution time can be deferred until such time as the facets are actually needed (i.e., by an application). In this way, some of the runtime and most of the space costs of supporting multiple versions are only spent when absolutely necessary.

The strategy of deferring the actual maintenance of a dependency constraint until its effect is actually required can be applied as well to the propagation of information among the facets of an instance. Rather than update the attribute values of the other facet(s) each time a facet attribute is modified, one need only bring a facet up-to-date when there is an attempt to access it. This scheme can be supported by associating a flag with each facet indicating whether the facet is up-to-date with respect to the most recently modified facet. The application of read methods on facets with an unset flag are preceded by a resynchronization operation, which performs any necessary updates and sets the flag.

This scheme reduces overall runtime expense, since the resynchronization step is not performed in concert with every update operation, as was previously the case. However, it does increase the potential cost of previously inexpensive read operations.

To this point, we have been very liberal with our allocation of space for instance representation. Although the lazy allocation of facets conserves some space in the short run, the disjoint union representation model implies that every instance of a versioned class will have a complete collection of facets. There are a few optimizations that could be performed to reduce space requirements.

The first space-saving improvement entails having each set of shared attributes occupy a single slot in the multifaceted representation. A performance improvement might also be realized here, since slot sharing reduces the expense and/or frequency of update propagations (Figure 5).

Under certain circumstances, the slot associated with a derived attribute can be recovered as well. If an inverse procedure to the derivation function is known to the system, then the attribute can be simulated by appropriate reader and writer

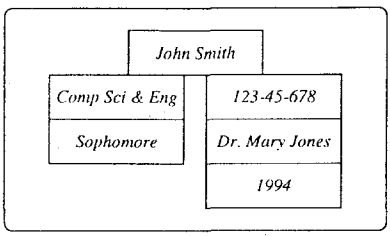


Figure 5. Multifaceted instance representation using common slot for shared attributes.

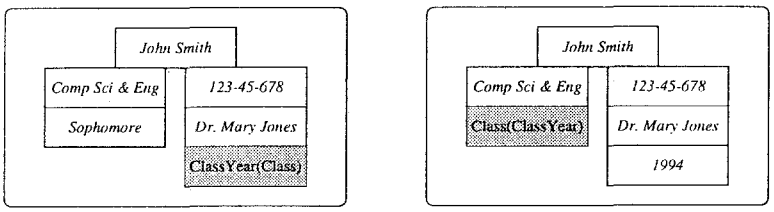


Figure 6. Multifaceted instance representation minimizing derived attribute allocation. For the `Undergraduate` class, two minimizations exist.

methods. For many evolutions, the inverse procedure appears as the derivation function for the related attribute in the other facet. The `Class` and `Class Year` attributes in our example are related in that way (Figure 6).

From a runtime performance perspective, this space optimization reduces the expense of write methods while making read methods more costly. The slot allocated for a derived attribute acts as a cache for its derivation function and, depending on the frequency of modifications to its dependent attributes in the other facet(s), its maintenance might be more time-efficient.

Note that the emulation scheme of the Encore system is an extreme case on the space vs. time spectrum. In the Encore system, however, emulation was the only option. In our scheme, the programmer could choose to completely emulate a facet in situations where time is less of a concern than space, and where all the attributes are derivable from other facets.

3.5. Subtyping

We have, to this point, failed to explain how our class evolution and instance adaptation scheme copes with our model's subtyping mechanism. We first identify

what characteristics of subtyping are problematic with respect to evolution and adaptation and then motivate our solution.

One of the characteristics of a class in our data model is that a class can be declared a subtype of one or more other classes. As a subtype, it is required to minimally export the interface of each of its supertypes. One possible evolution in such a model is a change to the supertype list. Such changes (i.e., addition or deletion of a supertype) involve only the addition and/or (optional) deletion of sets of methods in the new class version.

Unfortunately, problems supporting evolutions upon a superclass (i.e., a class with other classes that have been declared as subtypes) cannot be dismissed so easily. At issue is what to do with subclasses (declared subtypes) when a class is altered.

A subclass (subtype) is distinguished from an application or a class that depends on a particular class interface. For a subtype, backward compatibility is not altogether useful, since, in order to maintain the subtype relationship, it is obliged to evolve its type (interface and semantics) in concert with its supertypes.

If the new version resulting from an evolution is a superclass of the previous class-version, then the set of subclasses remain subtypes. However, if the evolution specializes or more drastically changes the class's interface or semantics, the subclasses will not be subtypes of the revised class without evolving them as well.

Note, however, that the subclasses *remain* as subtypes of the old version of the class. If we were to not allow the versioning of the subtyping hierarchy, a programmer would be obligated to evolve all the subclasses. Alas, it cannot be assumed that the database programmer performing the class evolution has the will or the means (due to the existence of multiple database programmers) to evolve an entire subtree "below" the evolved class. Therefore, allowing versioning of the type hierarchy (i.e., allowing classes to be subtypes of class-versions and not classes), appears to be the correct approach.

*3.5.1. **Distribution.*** When the database is distributed, one significant representation advantage might be the distribution of facets. Facets could be created and located at the sites where they were needed, as opposed to where their co-facets reside. Whether this could prove beneficial depends upon the relative frequency of write over read operations, and the degree of dependency among facets. When facet distribution is advantageous, one would want to optimize time over space, and directly represent shared and derived attributes. Such an approach would also improve fault-tolerance.

3.6. Customizing an adaptation strategy

Just as the actual specification of the dependency relationship between facets is specified by the programmer, certain other aspects of the adaptation should

be accessible to programmer control as well, including: whether compatibility is required, whether or not to maintain old facets, and the possibility of multiple active instance representations.

While important in general, program compatibility is not always required (e.g., a database with a single application program and a single user). In such situations one should be able to employ the minimally expensive strategy and instruct the system to convert existing instances fully and discard (or perhaps archive) the old information. Furthermore, conversion and compatibility are not mutually exclusive. As long as an inverse conversion procedure is known, one could convert and emulate the older interface. This might be useful when you want to preserve compatibility, but expect that it will be needed infrequently enough that you are willing to pay the cost of emulation in those instances. If an application tends to reference a distinct subset of the instance collection, one could employ a strategy that converts (on access) instances to the version of the application.[7] The important characteristic of this evolution architecture is that the database programmer has access to the control knobs and can tune the evolution strategy to improve performance.

Sometimes, modification of the database or its schema is impossible. Databases might be read-only for permission (e.g., remote database exported as a public service) or licensing reasons (e.g., reference materials on CD-ROM). In such situations, something resembling Zdonik's wrapping scheme must be used, with the wrapper actually residing in a separate database. The programmer must have a way to specify this situation to the system.

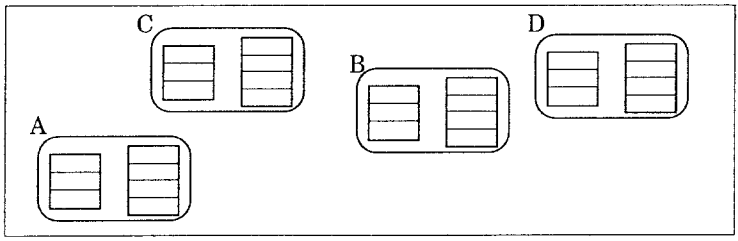
Often the access patterns are not particular to an entire class, but only to a subset of the instances. In such cases, additional efficiency could be achieved by employing different representations for differently accessed instances. Such functionality would obviously require extensive programmer influence, notably the inclusion of a procedure to determine which representation to employ. This procedure might be functional (e.g., depending on the state of the object) or require the maintenance of its own state (e.g., accounting information). Such advanced adaptation schemes will be the subject of a future paper.

3.7. Multifaceting and heterogeneity

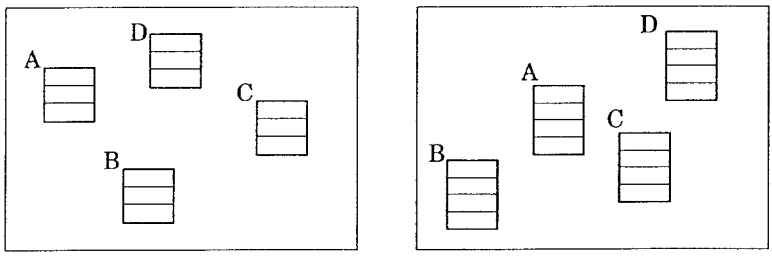
While we have made no explicit mention of it, our figures have depicted our multifaceted persistent instances as contiguously stored objects. Locality among facets is advantageous when the propagation of values is frequent. However, deferring the update propagation until the facet is actually referenced can reduce the benefits of facet colocality.

Instead of grouping facets by object, we could instead group them by class-version. Partitioning the database in this manner is reminiscent of the heterogeneous database systems introduced at the outset of this paper (Figure 7).

To support backward application compatibility across schema evolutions, we



(a)



(b)

Figure 7. Distribution of facets: Facets of the same objects could be distributed across schemas, resembling a heterogeneous distributed database.

have introduced schema heterogeneity. This reemphasizes the similarities between the schema evolution problem and the schema integration problem. Toward the end of the previous section, we stated that a general mechanism could be used to support both the evolution and heterogeneous schema integration processes. We intend to show that our "multifaceted" scheme is such a mechanism, but first we must discuss an important distinction between evolution and integration.

3.8. Schema integration revisited: the instance integration problem

Because evolution implies a migration from one schema to a new one, it can assume that the latter has no existing instances. However, the schemas that are part of integration procedure typically *do* have existing instances.

Why is this a problem?

Thomas et al. [32], identify four ways in which the objects in the two databases being integrated can relate to each other:[8]

Replication: when the objects in one database are copies of objects found in the other database

Vertical Fragmentation: when the objects are instances of a type found in the other database
Horizontal Fragmentation: when they are extensions of objects found in the other database
Data Mapping: when they are functional translations of objects found in the other database

Although developed for distributed RDBS, the four roles hold in the DOODBS context, with one caveat: as objects in the object-oriented data model possess an identity, objects linked via integration (i.e., objects fulfilling *replication*, *horizontal fragmentation*, or *data mapping* roles) much share an identity. The **instance integration problem** (sometimes called *object integration* or *identity integration* [5, 7, 18]) is concerned with collecting and linking together these related objects. We will address this issue presently.

3.9. Supporting schema integration

We have motivated schema versioning and application compatibility by observing that upgrading applications or databases is not always practical or possible. The member databases of a heterogeneous DOODBS exemplify our claim; different histories, requirements, or administrations have kept these schemas (and associated databases) separate in the past. Integration is a cooperative venture meant to simplify the development of applications requiring information from more than one of these databases. Integration need not mean assimilation; the members need not (and often cannot) surrender their autonomy/identity in the process.[9]

We can thus view integration (partly) as the simultaneous evolution of multiple schemas to a common interface, subject to the constraint that the individual components (i.e., schemas and representative instances) remain intact. Such a requirement is not an obstacle, since our evolution scheme supports the simultaneous existence of multiple schemas, and the simultaneous existence of multiple representations, in the form of multifaceted instances.[10]

At the conclusion of the integration process, our integration schema will be populated with such multifaceted instances, the facets remaining in their original databases (as illustrated in Figure 7). Only the instance integration problem remains.

For the (parts of the) schemas that are fragmented vertically, like the `Library Entry` class in our library integration example (Figure 1), no grouping is necessary, and we can assign unique (universal) identifiers to each object. However, if any other form of integration is present, we must assign the corresponding facets a common id.

As we are not able to rely on object identity, we require another way of identifying corresponding objects. This task is an active research problem, and

we will not present a definitive solution here. One basic approach is to identify corresponding objects by some common feature (e.g., ISSN or ISBN identification for books, personal name, etc.) [18]. Such common features would typically be the shared or derived attributes that exist between the class we are trying to integrate.

However, in any specific database, these common features may not sufficiently unique to adequately identify correspondences. Bratsberg [7] considers providing for user intervention in such situations.

4. Multifaceting and other advanced database features

In the course of our presentation, we have assumed the existence of a number of advanced OODBS features, and serendipitously implemented others.

4.1. Remote object references

The observation that the facets comprising a multifaceted object could be distributed assumes that the DOODBS supports nonlocal references to objects. As these potentially distributed facets share an identity and thus a universal identifier, the intraobject linkage could be implemented using the local database ID and the universal ID. However, some sort of facet-level, remote ID mechanism would facilitate the maintenance of interfacet constraints.

The heterogeneous schema integration mechanism described above relied on the existence of remote identifiers, so as to unify the distributed instances into an integrated object.

4.2. Object migration

By *object migration*, we mean the ability of objects to be moved from one database to another [16]. An object is migrated (either automatically or explicitly) to improve locality between itself and the objects it is related to (typically via reference pointers). Similarly, we might imagine *facet migration*, whereby facets might relocate to sites whence they are referenced. Note, however, that if interfacet dependencies require frequent maintenance, the distributed fragmentation of an object might not be beneficial.

The multifaceted approach to heterogeneity provides a mechanism for object migration for heterogeneous DOODBS. Moving an object across a schema barrier necessitates converting it. With multifaceting, the original structure can be retained and updated, as necessary. Thus, when an object migrates, it leaves a piece of its "soul" behind.

4.3. Replication

To improve performance and reliability, multiple copies of database stores are often maintained. At the simplest level, these *replication sites* provide benefits for read operations only (the write operations requiring updating to the all replication sites).

Although replication is often considered at the granularity of the database-file, replication at the object level could, to a large degree, be considered a special case of our multifaceted (and fragmented) instance scheme.[11] The shared slots of the instances of two (or more) class-versions are, effectively, replicated among the various distributed databases on which they (i.e., the instances) are active. Just as replicated databases must propagate side effects to their copies, the effects of a side effect to one facet of an object much be propagated (eventually) to its "co-facets." Full object replication could be implemented by duplicating schemas on other sites, and maintaining duplicate facets there.

The similarlity to replication extends to **lock management** as well. Locking an object requires locking *all* of its facets, whereever they may be located, just as all the replicated sites of an object must be locked for write operations [16].

5. Conclusions and future research

This paper describes a new, highly flexible approach to supporting schema evolution in object-oriented database systems. While not dependent on distribution, its support of arbitrary evolutions and application compatibility makes it attractive for use in the same type of application contexts that are appropriate for distributed OODBS. Schema versioning and instance multifaceting are the mechanisms by which compatibility are supported. By allowing the schema designer the ability to specify the precise relationship between class versions, a wider variety of evolutions than previous schemes can be supported.

The paper also highlighted the similarities between schema evolution and heterogeneous schema integration, and described how the aforementioned schema evolution support mechanism can also assist with the schema integration task.

Some minor contributions include the relationship between multifaceting and object replication, and the potential for facet migration.

We have left unresolved a number of issues. Some are addressed in [9], but most are topics for additional research. These issues include:

- Precisely how the programmer specifies the schema evolution and the adaptation strategy details: including dependency functions and representation decisions. Work has begun on a special-purpose language. One significant benefit of having a language is that common adaptations could be maintained in a library. The basic evolutions as specified in [2] could thus be provided in the form of library routines.

- How to support evolutions that involve more changes to more than one class. (The simplest of such evolutions is called *telescoping* [3, 9, 26].)
- How best to be able to match up integrating instances among heterogeneous schemas.

Notes

1. Although "schema" is already a plural noun, we will use the relatively popular "schemas" to refer to multiple, distinct schema.
2. Whether the instances are converted eagerly or lazily becomes an implementation issue.
3. A case could be made that since schema evolution is not a frequent occurrence, it would not be unreasonable to require that all constituent database servers be functioning at evolution time. However, as we see no major increase in implementation complexity associated with the introduction of fault-tolerant, distributed schema evolution, we address the issue at this time.
4. For explanatory purposes, imagine that we are describing a class consisting of only two versions, and where the database is populated by instances of both.
5. Common in the semantic sense, i.e., having the same type and meaning.
6. Independent attributes can be initialized using the default values from the regular class definition.
7. This is the approach taken by Monk's CLOSQL system [4]. See Section 2 for details.
8. The reduction from n to two databases is for illustrative purposes only, and does not affect our argument.
9. Similar to the member countries of the European Community.
10. We observe that four categories introduced by Thomas et al. [32] have analogues in our classification of the attributes between facets: replication is analogous to *shared attributes*, data-mapping to *derived attributes*, and horizontal fragmentation to *independent attributes*. Vertical fragmentation is implied by our schema versioning scheme, due to the fact that all instances of one class-version are also instances of all other class-versions. *Dependent attributes*, combining characteristics of independent and derived attributes, were not identified by the authors, probably because of the simplicity of the relational data model.
11. The size and complexity of some objects in an OODB could make replication at object granularity practical.

References

1. R.S. Arnold (ed.), *Tutorial on Software Restructuring*, Washington, DC: Institute of Electrical and Electronic Engineers, IEEE Society Press, 1986.

2. J. Banerjee, W. Kim, H-J. Kim, and H.F. Korth, "Semantics and implementation of schema evolution in object-oriented databases," in U. Dayal and I. Traiger (eds.), *Proc. of SIGMOD Int. Conf. Management of Data*, San Francisco, CA, May 1987.
3. E. Bertino, "A view mechanism for object-oriented databases," in *Advances in Database Technology–EDBT '92 Int. Conf. Extending Database Technology*, Vienna, Austria, February 1992, pp. 136–151.
4. A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object structure in emerald system," in *OOPSLA86* [28], pp. 78–86.
5. S.E. Bratsberg, "Integrating independently developed classes," in *Proc. Int. Workshop on Distributed Object Management*, Edmonton, Canada, August 1992.
6. S.E. Bratsberg, "Unified class evolution by object-oriented views," in *Proc. 11th Int. Conf. Entity-Relationship Approach*, October 1992.
7. S.E. Bratsberg, "*Evolution and Integration of Classes in Object-Oriented Databases*," PhD thesis, The Norwegian Institute of Technology, University of Trondheim, June 1993.
8. R. Bretl, D. Maier, A. Otis, J. Penney, B. Schuchardt, J. Stein, E.H. Williams, and M. Williams, "The GemStone data management system," in Won. Kim and Frederick H. Lochovsky (eds.), *Object-Oriented Concepts, Databases and Applications*, Reading, MA; Addison-Wesley, 1989, chapt. 12.
9. S.M. Clamen, "Class evolution and instance adaptation," Technical Report CMU-CS-92-133, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, June 1992.
10. C. Delcourt and R. Zicari, "The design of an integrity consistency checker (ICC) for an object oriented database system," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Lecture Notes in Computer Science*, vol. 512, Geneva, Switzerland, Springer-Verlag, July 1991. A more detailed version is available as [11].
11. C. Delcourt and R. Zicari, "The design of an integrity consistency checker (ICC) for an object oriented database system," Dipartimento di Elettronica Technical Report 91.021, Politecnico di Milano, Milan, Italy, 1991. A short version of this paper appears in the 1991 ECOOP proceedings.
12. D. Giuse, "Kr: Constraint-based knowledge representation," Technical Report CMU-CS-89-142, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, April 1989.
13. W.G. Griswold and D. Notkin, "Program restructuring to aid software maintenance," Technical Report 90-08-05, Dept. of Computer Science and Engineering, University of Washington, Seattle, WA, September 1990.
14. A. Nico Habermann et al., "Programming with views," Technical Report CMU-CS-TR-177, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, 1988.
15. S. Heiler, U. Dayal, J. Orenstein, and S. Radke-Sproull, "An object-oriented approach to data management: Why design databases need it," in *Proc. 14th ACM/IEEE Design Automation Conf.*, pp. 335–340, January 1987.
16. E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-grained mobility in the Emerald system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 109–133, February 1988.
17. S.E. Keene, *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, Reading, MA, 1989.
18. W. Kent, "The breakdown of the information model in multi-database systems," *SIGMOD Record*, vol. 20, no. 4, pp. 10–15, December 1991.
19. S.N. Khoshafian and G.P. Copeland, "Object identity," in OOPSLA86 [28], pp. 406–416.
20. W. Kim, J.F. Garza, N. Ballou, and D. Woelk, "Architecture of the orion next-generation database system," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 1, pp. 109–24, March 1990.
21. W. Kim, *Introduction to Object-Oriented Databases*, Computer Systems, Cambridge, MA, MIT Press, 1990.
22. W. Kim, N. Ballou, J.F. Garza, and D. Woelk, "A distributed object-oriented database system supporting shared and private databases," *ACM Transactions on Information Systems*, vol. 9, no. 1, pp. 31–51, January 1991.

23. B.S. Lerner and A.N. Habermann, "Beyond schema evolution to database reorganization," in *Proc. ACM Conf. Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA) and Proc. European Conf. Object-Oriented Programming (ECOOP)*, Ottawa, Canada, October 1990, pp. 67–76. Published as ACM SIGPLAN Notices 25(10).
24. B. Liskov, "Data abstraction and hierarchy," in *Proc. ACM Conf. Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 17–34 (addendum), Orlando, FL, September 1987.
25. S. Monk and I. Sommerville, "A model for versioning classes in object-oriented databases," in P.M.D. Gray and R.J. Lucas (eds.), *Proc. Tenth British National Conf. Databases (BNCOD10), Lecture Notes in Computer Science*, vol. 618, pp. 42–58, Aberdeen, Scotland, July 1992. Springer-Verlag.
26. A. Motro, "Superviews: Virtual integration of multiple databases," *IEEE Transactions on Software Engineering*, vol. 13, no. 7, pp. 785–98, July 1987.
27. M.T. Özsu and P. Valduriez, *Principles of Distributed Database Systems*, Prentice-Hall, 1991.
28. *Proc. of the ACM Conf. on Objected-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, Portland, OR, September 1986.
29. A.H. Skarra and S.B. Zdonik, "Type evolution in an object-oriented database," in *Research Directions in Object-Oriented Programming*, MIT Press Series in Computer Systems, MIT Press, Cambridge, MA, 1987, pp. 393–415. *An early version of this paper appears in the OOPSLA '86 proceedings.*
30. G.L. Steele, Jr., *Common Lisp: The Language*, 2nd ed., Digital Press, 1990.
31. M. Stonebraker, "Implementation of integrity constraints and views by query modification," in *Proc. SIGMOD Int. Conf. Management of Data*, San Jose, CA, 1975.
32. G. Thomas, G.R. Thompson, C.W. Chung, E. Barkmeyer, F. Carter, M. Templeton, S. Fox, and B. Hartman, "Heterogeneous distributed database systems for production use," *ACM Computing Surveys*, vol. 22, no. 3, pp. 237–266, September 1990.
33. E. Waller, "Schema updates and consistency," in *Proc. Second Int. Conf. Deductive and Object-Oriented Databases*, 1991, pp. 167–188.
34. S. Zdonik, "Object-oriented type evolution," in *Advances in Database Programming Languages*, Francois Bancilhon and Peter Buneman (eds.), ACM Press, New York, NY, 1990, pp. 277–288.
35. R. Zicari, "A framework for schema updates in an object-oriented database system," in *Building an Object-Oriented Database System: The Story of O_2*, Morgan Kaufmann, 1992. Also available as Politecnico di Milano, Research Report no. 90-025.
36. Roberto Zicari, "A framework for o_2 schema updates," Rapport Technique 38-89, GIP Altaïr, Rocquencourt, France, 1989.