Research issues in database schema evolution: the road not taken


Sudha Ram
Department of MIS
The University of Arizona

G. Shankaranarayanan1
Information Systems Department
Boston University School of Management
gshankar@bu.edu
Phone: (617) 353-4605
Fax: (617) 353-5003

1 Boston University School of Management, Department of Information Systems, 595 Commonwealth Avenue, Boston MA 02215. Email: {gshankar@acs.bu.edu}, Phone: (617) 353 4605 Fax: (617) 353 5003
Abstract

Dynamic schema evolution is the ability of the database schema to evolve by incorporating changes to its structure without loss of existing data and without significantly affecting the day-to-day operations of the database. A very large body of literature exists today reflecting the extensive work addressing schema evolution and its management. Schema evolution has three well-defined and inter-related activities: core schema evolution, version management, and application management. These activities have been examined using different databases and associated data models – the object-oriented, relational, and semantic data models. The first objective of this paper is to examine schema evolution research and present a comprehensive summary of the different research activities from both perspectives. Researchers have examined schema evolution in single stand-alone databases and schema evolution in an integrated heterogeneous set of databases has received little attention. Such systems are common in organizations either as part of large information systems or in data warehouses. The second objective is to highlight the importance of schema evolution in heterogeneous database systems. Managing schema evolution in heterogeneous database systems is much more complicated than in single standalone databases and may therefore require automated support. The third objective is to critically examine the implications for automating schema evolution.

Categories and Subject descriptors: Database management, logical (conceptual) design, database administration, heterogeneous databases, and information systems.

General Terms: Data models, database schema, data dictionary/directory, data translation, and decision-support systems.

Key Words and Phrases: Database schema evolution, dynamic schema evolution, object-oriented databases, relational databases, semantic data models, heterogeneous database systems, semantics of schema changes, propagation of schema changes, and metadata management.
1. Introduction

A database typically represents a view of some real world referred to as the world of reference of that database. A database schema is defined as the description of the database [Elmasri and Navathe 1994] and as the overall design of the database [Silberschatz et al. 1997]. A database schema describes the structure of the database and models the objects in its world of reference. Defining the database schema is an evolutionary activity requiring more than a single iteration to capture the complex world of reference represented by the database. Even after defining it, the database schema may continue to change and evolve [Zanilo et al. 1997]. When the world of reference of the database changes the schema must change to be consistent with its world of reference. User requirements change/increase over time and the database schema must change to support these changed/new requirements. The changes hence continue to occur even when the database is operational.

The extent to which changes occur to a database schema has been well illustrated in the study conducted by Sjoberg [Sjoberg 1993]. This study is based on a health management system using a single relational database and the changes were studied during the development and operational phases of the system. During the development phase (5 months) there were 65 changes (additions and deletions) to relations and 470 changes (additions and deletions) to attributes. During the operational phase (11 months) the corresponding numbers were 299 and 2324 respectively. This study has brought out an important fact about schema evolution: schema changes are as common during the operational phase of the database as it is during the developmental phase. The impact of schema changes has also been illustrated in many other application domains in organizations. Theses include office information systems [Gibbs and Tsichritzis 1983], [King and McLeod 1985], personal databases [Cattel 1983], [Lyngbaek and
McLeod 1984], design engineering databases [Afsarmanesh et al. 1985], [Batory and Kim 1985],
and artificial intelligence systems [Kerschberg 1984], [Stefik and Bobrow 1984].

A database schema needs to change due to one or more of the following reasons: (1) Users do not know in advance (or are not able to specify) the desired functionality of a large application system and hence their perceptions and requirements change over time. The initial schema needs to change to accommodate the changing perceptions of the users. (2) The world of reference of the database is continually changing. The database schema must change to be consistent with it. There is yet another reason: databases design is more of an art than science and when performed by a designer unfamiliar with the domain results in a database that is simply incorrect [Borgida and Williamson 1985]. This can be treated as an extreme case under (1) were user requirements were captured (very) inaccurately. Consider for example a database called the University-Courses (U-C) database that currently stores data on the different courses offered in a (any) department in some university. A database schema modeled using the ERM is shown in figure 1. The components and relationships that exist in this real world include courses offered in the department, the Professors that teach these courses, and Students who enroll in the courses.

**Figures 1 and 2: Database Schema Model for Course Information Example – Before and After Change**

**Changes to the real world:** It may be necessary to expand the world of reference to include the entire school (engineering, business, or arts and science) instead of just one department. Since there are many different departments it is now necessary to include data on departments.
Professors are usually associated with one department and the database should capture this relationship. The students may each have a major department and so students must be related to departments. The schema must change to reflect the new structure shown in figure 2. The data must also change to conform to the new schema (new data may be added and data corresponding to relationships between the instances defined or redefined).

**Changes to Users' perception of the real world:** Administrative assistants need information on graduate students supported by their department. Since the database has a single collection of students, it may now be necessary to classify students as graduate and undergraduate. It is necessary to add new information that classifies the students into the two categories. The semantic model of the changed schema is shown in figure 2.

![Figure 3: Database Schema Model for Course Information Example: After Classifying Students](image)

These changes transform a database schema from an existing state to a modified new state. It is important these changes are managed to maintain the database consistent with its world of reference and accommodate user requirements. The data in a database is dependent on the schema of that database. When the schema changes, the data needs to be changed to “match” the changed schema. Therefore managing schema changes also implies managing changes to the associated data. Schema evolution may be defined as the ability of the database to respond to the
above changes without loss of the existing data and without significantly affecting the day-to-day operations of the database.

Reorganizing the database has been the solution to manage schema evolution. This is performed infrequently because during reorganization, part or all of a database is inaccessible to users. During the periods between reorganization, the consistency of the database with its world of reference is not guaranteed. Research addressing online reorganization of relational databases attempts to solve problems such as restoration of clustering, purging old data, creating back up copies, and online index creation [Sockut and Iyer 1996]. Sybase System 11 provides support for on-line database consistency checking and backup/restore operations [Rengarajan et al. 1996]. Algorithms for on-line clustering [Omiecinski 1996], and for on-line reorganization of the \( B^+ \) trees used for clustering indexes in relational systems have been proposed [Zou and Salzberg 1996]. These address performance of online reorganization and not schema evolution. To ensure continuous access to data while incorporating schema changes the process must be managed dynamically. Dynamic schema evolution is defined as managing schema changes in a timely manner without loss of existing data while the database continues to be operational and without significantly impacting the day-to-day operations of the database.

Changing the database schema has a number of implications. First, a change to one part of the schema may require more changes to other parts of the same schema (cascading changes). All of these changes must be identified and incorporated. Second, when incorporating changes to the schema, the consistent state of the schema must be preserved. Procedures to define a consistent state and mechanisms to preserve it must be defined. Third, the changes to the schema must be translated or propagated to the data in its corresponding database. When propagating changes, existing data in the schema may be lost. Also, when the schema is changed, external
application programs that are part of the information system will no longer be able to access the changed schema. To preserve existing data and support existing applications multiple versions of the schema must be maintained. Versions must be managed so that they are correct and consistent. Based on these requirements three inter-related activities can be identified in schema evolution. **Managing core schema evolution** includes identifying and incorporating changes to the schema while preserving the consistent state of the schema as well as propagating the changes to the data associated with the schema. The issues that need to be addressed for managing core schema evolution are:

- Understanding all possible changes to the database schema
- Understanding the implications of each change
- Incorporating changes to an existing schema while ensuring that the consistent (and correct) state of the schema is maintained.
- Determining how a change (may/may not) affects other parts of that schema
- Propagating changes to the data associated with the changed schema so that the data is consistent with the changed schema.
- Performing these changes dynamically without significantly impacting day-to-day operations of the database.

The management of changes to the schema is termed as semantics of schema changes [Banerjee et al. 1986][Peters and Özsu 1997] and as schema evolution [Zanilo et al. 1997]. Propagating the changes to the data in the database is termed as database evolution [Zanilo et al. 1997] and as propagation of schema changes [Banerjee et al. 1986][Peters and Özsö 1997]. Core schema evolution includes both schema evolution and database evolution.

The second research activity is **version management**. This deals with the management of the different versions of a database schema introduced by schema changes. Evolutionary changes to the schema may require retention of past states of the schema and managing multiple versions of the schema is essential. Version management must address the following:
• Defining the granularity of versioning (entire schema or just schema-objects)
• Identifying versions of the schema and associating each version with its parent (the base from which a version is created)
• Maintaining a mapping between available versions (what is different?)
• Conforming the existing data to a new version of the schema and associating the data with its corresponding schema version.
• Specifying the current or active (or default) version of the schema

Management of applications is the third research activity and it examines how dependent applications may continue to work when the schema is changed. Changes to the structure often result in existing applications becoming obsolete. Managing applications so that these might access the data despite changes to the database schema is the focus here. Management of applications is closely related to managing multiple schema versions. An application would continue to work if the schema version it is associated with continues to be accessible. Typical activities in application management include:

• Creating an interface for each schema version. As the interface identifies the set of all objects available in a given version applications must interact with a versions only via its interface.
• Providing mechanisms associated with each version interface for translating data (instances) between parent and child versions.

Schema changes can be identified based on a specific type of database (or a specific data model). Schema changes in an object-oriented database are different compared to changes in a relational schema and database. Based on this observation research in schema evolution can be classified based on the data model used. The majority of research in schema evolution is based on object-oriented databases and the associated object data model. This can be attributed to the fact that schema evolution may have had its origins in engineering design databases that are predominantly object-oriented databases. Relational data model has also been used though to a lesser extent. Besides these two semantic and other data models have also been studied.
Consider the U-C example described in section 1.1. The semantic model representation of the schema(s) is shown in figures 1 and 2 and the class hierarchy representation for the same is shown in figure 4. If a relational model were used to represent the schema, the original schema would have relations for Professors, Students, Courses, and relations for the Teach and Enroll relationships (assuming that the cardinality is many-many for each). Adding Departments triggers a change to the current schema.

**Figure 4: Class-hierarchy for the original and changed U-C database schema**

Let us examine the implications of this schema change from the point of each of the research directions described earlier. Considering core schema evolution, depending on the data model used to describe the schema of the U-C database, a new relation (or a new object class or a new entity class) needs to be defined in the existing schema to hold the data on departments. This would also mean that a set of attributes must be defined for the relation (or object/entity class) based on the data that needs to be stored. Adding department information will trigger some more changes to the schema. The new relation (object/entity) that is added must be related to one or more of the existing relations (objects/entities), Professors and Students in this case. These are relationships that exist in the real world and representing them in the schema is necessary to maintain the database consistent with its world of reference. Incorporating these relationships
may introduce new relations and/or extend existing relations in a relational model. In object models, these object associations are captured by defining methods (or behaviors) in one or more objects. If a semantic model is used to represent the schema these associations are captured by relationships between one or more entity classes. To make the above changes, operators (or functions) that help define new relations/objects/entities and new associations are needed. These operators must implement the changes dynamically and while doing so, must maintain the schema in a correct and consistent state. A framework that defines what is correct and consistent is needed to assist the operators implement schema changes. To complete the core schema evolution process, the data (or instances) must be changed to conform to the changed schema.

There may exist a set of database users who prefer (and demand) to use the original schema and data. To satisfy their needs and satisfy a different set of users that need the modified data (and its schema) simultaneously it is necessary to maintain both the original and the changed schema versions. Each version must be uniquely identified. One must also decide whether to maintain two independent versions of the schema, one with the department information (new version) and one without (old version). In some instances, it may be sufficient to maintain the incremental changes only as part of the new version along with the information that defines how this links to the old version. Yet another way of maintaining the two versions is to define the old version as a “view” of the new version. In this example, the old version would be a subset of the new version. The roles may be reversed if the change was a deletion instead of an addition. An important question arises when another schema change (classifying students as shown in figure 3) is needed. Should the change be implemented based on the old version or the new? In the former case, the old version is the parent of the third version created and in the latter case it is the second. A more general method would define an all-encompassing schema that
includes every relation/object/entity that ever existed, and define each version as a view of this
schema. Version management would have to address all of the above requirements.

A set of application programs that were developed based on the old U-C schema will
continue to work with the new schema. The addition of the department information does not
affect these applications as Professor, Student, and Course information exists in both versions.
On the other hand, if some information (say, Professors) were to be deleted, this would affect at
least some of the applications in the set. Also, to retrieve the new information (about departments
and departments associated with Professors, Courses, and Students) added to the database, a new
set of applications must be created. This new set will be incompatible with the old version of the
schema. So applications must be associated with versions or users must specify which version to
access when submitting applications. Application management in schema evolution attempts to
address these issues. An important reason for maintaining multiple versions might be to support
the dependent applications besides satisfying user demands.

Schema evolution has been examined using a single, standalone database only. Managing
schema evolution in a heterogeneous database system (or environment) has received little or no
attention. Why is this important? Organizations today implement comprehensive information
systems that include more than just one data source. Databases, of different types, are integrated
together to provide the necessary storage support. Each database schema in this set of multiple
databases may (and do) undergo changes and evolve due to reasons described earlier. The impact
of schema evolution in this heterogeneous collection of databases is much stronger than in any
single standalone database. The number of changes resulting from a single change is much larger
because the impact is not restricted to that one database where a change is first identified. Due to
the extensive inter-relationships between these heterogeneous databases a schema change,
besides affecting other parts of that schema, has the potential to affect parts of other database schemas as well. Identifying the changes completely including all cascading changes and propagating each schema change to its corresponding data (in the database) adds complexity. If one or more of these databases are distributed (as they often are), data associated with the changed schema object may be distributed across multiple locations. Each of these locations has to be identified and the changes must be made at all these multiple locations to maintain consistency. More than one database administrator (DBA) may be responsible for managing heterogeneous database systems. Each administrator is often responsible for one or more subsystem in the heterogeneous set. It is very difficult, and often impossible, for a database administrator to envisage the impact of a schema change and the many cascading schema changes that result. The difficulties arise not only in identifying all the possible changes but also in manually incorporating the changes. To ensure continuous access to data the schema changes must be managed dynamically. Hence dynamic schema evolution in heterogeneous database systems is an important research agenda and deserves a closer examination. Sections 2, 3, and 4 summarize schema evolution research based on the activities in schema evolution – core schema evolution (section 2), version management (section 3) and application management (section 4). Within each section, the organization follows the second perspective based on the data model used. The merits and drawbacks of each research are discussed. A schematic summary of the research examined here is shown in figure 5. Section 5 evaluates the requirements for managing dynamic schema evolution in a heterogeneous database system and a set of potential applications for dynamic schema evolution is described in section 6.
Object Model – focus on core schema evolution

Object Model – focus on schema evolution and version management (implies application management as well)

Object Model – focus on version control

Object Model – focus on application management

Relational Model – focus on core schema evolution

Relational Model – focus on version management (implies application management as well)

Relational Model – focus on schema evolution and application management

Relational Model – focus on schema evolution with version management (implies application management as well)

Conceptual Model – includes focus on schema evolution for versioning and application management

Restructuring a hierarchical database

* – applied to a relational database using ER (conceptual) model

** – applied to object-oriented databases using the ERM

Figure 5: Schematic representation of research reviewed
2. Core Schema Evolution

Management of core schema evolution involves identifying changes to the schema, incorporating the changes into the existing schema while maintaining the correctness and consistency of the schema, and reflecting these changes in the underlying set of database(s).

2.1 Core Schema Evolution and Object-Oriented Data Model

Object-oriented models offer some advantages for schema evolution when compared with relational data models. In an object-oriented model all conceptual entities are modeled using the single concept of an object. The state of the object is captured by the instances of the object and the behavior captured by the messages to which the object responds. The hierarchy of classes in object-oriented models defines properties of class hierarchies and inheritance. Objects in object-oriented models are represented using a lattice structure, known as the class lattice (for class objects and type lattice for type objects). The lattice provides a convenient way to determine the implications of the changes to itself and hence to the object schema. Thus the propagation of changes using the lattice structure is easier. Moreover, the object-oriented data model (and Object-Oriented Databases) is capable of modeling more details such as complex hierarchical objects and relationships about the real world compared to the relational data model.

In examining schema changes to the object model, one set of solution methods address the evolution of specific objects within the model such as class, type, or method objects [Penny and Stein 1987], [Skarra and Zdonik 1986], [Björnerstedt and Britts 1988], [Narayanaswamy and Rao 1988], [Osborn 1989], [Bertino 1992], [Bratsberg 1992], [Monk and Sommerville 1993], [Morsi et al. 1994], [Li and McLeod 1994], [Liu et al. 1997], [Lerner 2000]. Others address

---

2 The terms class and type are used interchangeably in a majority of the research discussed here. A class (or type) is a template for all its instances [Zdonik and Maier, 1990]. However some literature use the two terms to mean different things. A type refers to the specifications (or definitions) while the term class is used to represent the collection of current instances of that type [Zdonik and Maier, 1990].

Methods to propagate schema changes to the database include conversion on one side, and screening, filtering, or a hybrid on the other. Conversion coerces the existing data in the database to immediately conform to the changed / evolved schema. There is only one version of the database and schema available at any time. Data associated with an object that is no longer part of the new/evolved schema is not accessible from this schema and is “lost”. Screening and filtering methods for propagating schema changes are employed when multiple versions of the schema and data are maintained. Screening creates a new version of the schema along with a set of associated functions. When an application accesses this version of the schema, the functions coerce the data to conform to this schema version (delayed conversion). Filtering is similar to screening except it does not coerce the data. Instead the functions or filters associated with each version map the data to conform to that version.

Penny and Stein describe a method for class evolution in object-oriented databases by converting the existing instances to the modified version of the class in their prototype called GEMStone [Penny and Stein 1987]. GEMStone addresses the issue of modifications to the class and class definitions. It attempts to coerce the underlying database to conform to the new class definition adopting the early conversion method. GEMStone also uses a class hierarchy graph. Unlike the class lattice in ORION [Banerjee and Kim 1987], the graph in GEMStone is a tree where each class node has a unique super class. The consistency and correctness of this graph is
maintained by a set of six invariants and eight schema changes are defined based on the class-object. GEMStone does not version the class and maintains only the evolved or new class.

<table>
<thead>
<tr>
<th>Authors (and systems)</th>
<th>Method supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Penny and Stein, 1987 (GEMStone)</td>
<td>Class evolution</td>
</tr>
<tr>
<td>Skarra and Zdonik, 1986 (ENCORE)</td>
<td>Type evolution with version support</td>
</tr>
<tr>
<td>Björnerstedt and Britts, 1988 (AVANCE)</td>
<td>Class evolution with version support</td>
</tr>
<tr>
<td>Narayanaswamy and Rao, 1988</td>
<td>Instance inheritance augmented to SGODM, addresses “version problem” (only)</td>
</tr>
<tr>
<td>Osborn, 1989</td>
<td>Polymorphism in queries - addition of attributes of an aggregate object and addition of subclasses to an aggregate object</td>
</tr>
<tr>
<td>Bertino, 1992</td>
<td>Evolution of class objects using view mechanism with version support</td>
</tr>
<tr>
<td>Bratsberg, 1992</td>
<td>Generalization and specialization of class objects</td>
</tr>
<tr>
<td>Monk and Sommerville, 1993</td>
<td>Class evolution with version support</td>
</tr>
<tr>
<td>(CLOSQL)</td>
<td></td>
</tr>
<tr>
<td>Morisi, Navathe, and Shilling, 1994</td>
<td>Method evolution only</td>
</tr>
<tr>
<td>Liu, Zicari, Hursch, and Lieberherr, 1997</td>
<td>Polymorphism – method evolution</td>
</tr>
<tr>
<td>Lerner 2000 (TESS)</td>
<td>Compound Type Evolution</td>
</tr>
<tr>
<td>Banerjee and Kim, 1987, Banerjee et al. 1987 (ORION)</td>
<td>Evolution of all schema objects</td>
</tr>
<tr>
<td>Ngyuen and Reiu 1989</td>
<td>Evolution of all schema objects</td>
</tr>
<tr>
<td>Lerner and Habermann, 1990 (OTGen)</td>
<td>Evolution of entire schema with automatic database conversion</td>
</tr>
<tr>
<td>Zicari, 1991 (proposed methods for O²)</td>
<td>Evolution of all schema objects</td>
</tr>
<tr>
<td>Clamen, 1994</td>
<td>Evolution of all schema objects with version support</td>
</tr>
<tr>
<td>Breche, Ferrandina and Kuklok, 1995 (GOODSTEP, extending O²)</td>
<td>Evolution of all schema objects using view mechanisms</td>
</tr>
<tr>
<td>Ra, Rundenstiener, 1997</td>
<td>Evolution of all schema objects, transparent evolution using views</td>
</tr>
<tr>
<td>Peters and Őszu, 1995, 1997</td>
<td>Axiomatic Model</td>
</tr>
</tbody>
</table>

Table 1: Summary of Core Schema Evolution using Object-Oriented Model

Skarra and Zdonik examine the problem of schema evolution in object-oriented databases by versioning the type (different from class) of the object [Skarra and Zdonik 1986]. They take the view that objects in a database are shared and are persistent. However, the type definitions of

---

16
the object do change and the solution method addresses type evolution in object databases, utilizing the type lattice structure. The changes to the type include addition or deletion of a type, or modifications to the type definitions. In modifying a type definition, the properties, constraints, and operations that are defined by that type may be added, deleted, or changed. According to the authors, change to a type affects two groups of objects: objects of that type and its subtypes, and the program objects that use this type. An object takes on the properties and operations of its super type and hence programs that use super types of the changed type may also be affected. The first group of objects affected is handled by a version set mechanism. A new version of the type and subtype is generated whenever a type changes. All instances remain bound to one version and may be coerced to another version using screening. This methodology forms the basis of a prototype system, ENCORE, for type evolution.

AVANCE, another prototype system for schema evolution in object-oriented databases proposes general object versioning as the solution method [Björnerstedt and Britts 1988]. Object versioning is extended to support class (or type) versioning. New version of the class is created following a change to the class. Changes to class object include changes to attributes or properties, adding and deleting classes, and defining and removing subclasses. Changes to methods are not addressed. Change propagation to databases is performed using a method similar to screening.

In engineering domains, one specific schema change is the "version problem". An object that is originally defined as an instance of a particular type now evolves into a type in itself. Narayanaswamy and Rao discuss a solution method to this problem using a simple, generic, object-oriented data model (SGODM) [Narayanaswamy and Rao 1988]. If an instance \( I \) of type \( T \) evolve to be a type in itself the solution using SGODM would be to create a new subtype \( ST \) that
is made a subtype of $T$. $ST$ assumes the properties of $I$. Variations of $I$ that triggered the change are made instances of $ST$. This method of incorporating the change affects the application programs that operate on the object schema. The solution method proposed augments SGODM with "instance inheritance" that solves this problem without introducing new subtypes. A new link, "has-version", is introduced into SGODM. This is a multi-valued attribute link that links an object to all its versions. Using this link, instance $I$ can be linked to all its variations. Thus the original type-lattice structure remains unchanged (except for the addition of the new link(s)) and the application programs may continue to access the original lattice, which is preserved. The change is incrementally modeled in the schema by the introduction of this new abstraction. The proposed solution thus transforms an object schema and does conclusively state if multiple versions are needed and if so, how many versions to maintain. Method(s) for propagating changes to the data instances are not explicitly addressed.

Osborn presents an approach to schema evolution in object-oriented systems using polymorphic object algebra [Osborn 1989]. When the schema of an object database changes, the database is changed to conform to the changed schema. The notion of polymorphism is that certain queries or applications will return identical or equivalent results when applied to object databases, both before and after the change. Such polymorphic queries would be transparent to schema changes. Two types of changes are examined, modifying an attribute to an aggregate object (an object created by juxtaposing other objects of different types), and adding a set of subclasses to an aggregate object in a generalization hierarchy.

Bertino presents a schema evolution language that is an object-oriented adaptation of the view mechanism in relational database systems [Bertino 1992]. In particular, the definition language proposed permits definition of views that augment class definitions. This research
supports inheritance and object identifiers (OIDs) for view instances, two important characteristics of object-oriented data models and databases that are not seen in relational models and relational databases. View instances with object identifiers are physically realized in the database. This enables the view mechanism to support schema changes that specify the addition of attribute(s). In addition, the view mechanism has been extended to support schema versions and experiment with schema changes before realizing each. In this research, views are also used to define dynamic sets and partitions of classes, without affecting the class hierarchy organization. Two issues are not explicitly addressed here: instance adaptation (change propagation) and compatibility between versions.

Another research that implements object-oriented views for schema evolution in object-oriented systems is described in [Bratsberg 1992]. It supports changes to class objects like generalization and specialization, similar to the Bertino’s work. In addition, it also covers type versioning and class versioning to provide historical support for schema evolution. The class object is said to consist of its intent, a named set of properties, and its extent, a special set of objects. The intent is connected to an extent that has the same name as the intent. The extent is defined as being the set of objects created from the intent. The association between the intent of every object in the database is modeled as a graph called intent inheritance graph and the association between the extents modeled as the extent graph. Evolutionary changes to classes are defined in terms of intent and extent evolution. Propagation of changes is not addressed.

CLOSQL adopts the class (or type) versioning method for schema evolution [Monk and Sommerville 1993]. Multiple versions of a class may exist and CLOSQL employs a conversion method for change propagation. Instances are converted when there is a version conflict. The class version to be used is determined by the program that tries to extract the instances from the
system. The only changes addressed are the class-definitions and this research is dealt with in more detail under version management (section 3).

Morsi et al. describe schema evolution operations exclusively for the evolution of methods in object-oriented systems [Morsi et al. 1994]. This research supports addition, deletion, and modification of methods. It also supports certain schema change operations that would trigger changes to methods, such as renaming of classes, instance variables, and methods. The operations are captured in the prototype system GOOSE (Graphical Object Oriented Schema Environment). The schema information in GOOSE is maintained as a set of system objects belonging to system defined classes. Besides the fact that it is one of the few research works that address method evolution, GOOSE also provides methods for resolving inconsistencies that may arise during the evolution process. GOOSE maintains inconsistencies that arise during method evolution in system defined object classes and provides schema change operations to restore consistency by resolving the inconsistencies.

Li has described a system for schema evolution that incorporates machine-learning techniques [Li and McLeod 1994]. The prototype system, Personal Knowledge Manager (PKM), views evolution of the object schema in from the perspective of the evolution of object flavors. Object flavor is termed as the definition of the semantics of an object. A change to the fundamental semantics of the object is object flavor evolution. Every object in the database of PKM is related to another object using a mapping. A three-tier object flavor graph is defined to manage evolution of objects. The nodes in this graph are the different object flavors (atomic in tier 1, open atomic, closed atomic, social, mapping, and procedural in tier 2, open social set, closed social set, compound mapping, and compound procedural in tier 3). The edges that link the nodes define the evolutionary paths for object flavor evolution. A set of operators (procedural
objects) is defined to create and evolve object based on this graph. PKM is an end-user database evolution tool that attempts to automate the evolution process by learning from the user. The manner in which the evolution is reflected in the data is not clear.

Liu et al. describe the role of polymorphism and reuse mechanisms based on polymorphism for improving the adaptive capabilities of database application programs in an object-oriented environment [Liu et al. 1997]. The schema changes examined in this research are the changes to the behavior or method associated with the object classes. This is discussed further is section 4.1 under application management.

Lerner presents a model for type (or class) evolution in object databases [Lerner 2000]. Specifically, this research examines compound changes (moving or replacing attributes) to types that have thus far been treated as a sequence of simple changes. For example, moving an attribute from one type to another would typically be executed as a sequence of adding the attribute to one type and deleting it from another. However, the problem is with changing the corresponding data (or instances). Default instance values are assigned to the new addition and deletion results in all the old data values being lost. Such compound changes to types are supported by the model for type evolution presented in this research in which the attribute and its associated values are moved from one type to the other. It is important to note here that the maintainer (or administrator) needs to decide which operation (sequence of add/delete or move) is more appropriate for the specific change under consideration. The model is supplemented with a set of algorithms that compare the two or more types involved in the change and with the help of the maintainer, identify whether simple sequential changes are adequate or if compound changes are needed. The model is implemented in a system TESS (Type Evolution Software
System) that is then evaluated for its accuracy in detecting type similarities. The model (and system) does not support versioning.

The above discuss methods for incorporating changes to the object schema by identifying a set (or subset) of changes to one or more specific objects in the schema. The consistent state of the schema is defined using a formal structure such as a graph (the class lattice or object-flavor model) and this structure assists in determining operators and mechanisms for incorporating the changes. The propagation of these changes to the data instances is by conversion, or screening. Using views does not explicitly address how the data is changed but it is easy to envisage a method similar to filtering that maps the data for each view. Whether the evolution is dynamically managed or not is not explicit except in [Li and McLeod 1994]. Further, automating the evolution is addressed by [Li and McLeod 1994] and by [Lerner 2000] who point out that administrators/users must be involved in the process. When using polymorphism to support schema evolution, the thrust appears to be in the management of application programs. Hence polymorphism in schema evolution has been included here for completeness and the details of implementation are described further under the section 4 (application management).

ORION is a prototype object-oriented system that provides support for schema evolution [Banerjee et al. 1987a], [Banerjee et al. 1987b]. An important part of this work is the development of a formal taxonomy of schema changes and a framework for managing schema changes in object-oriented systems. The semantics of each schema change is examined and a set of invariant properties of the object-oriented schema is proposed which must be preserved across schema changes. In ORION, the authors describe an implementation method for schema evolution that does not require database re-organization or shut down. In this research, an interesting property of object-oriented data model has been brought to light. The class (or type)
lattice structure is treated as a graph (PIG or the Property Inheritance Graph), which has a system generated class called Object as its root. It is a rooted, directed, acyclic graph. The taxonomy, invariant properties, and operators for implementing schema changes are described based on this graph. The taxonomy is listed under three major classifications: changes to a node in the PIG, changes to an edge in the PIG, and changes to the contents of a node. The final category includes changes to the attribute of an object and changes to the methods defined on the object. A significant contribution of this work is that it provides a clear and systematic framework for managing schema evolution. Change propagation to the database is by screening.

Nguyen and Rieu present a set of schema changes similar to that in Orion [Nguyen and Rieu 1989]. While the semantics for schema changes is similar to Orion, the user is permitted to decide between screening and conversion for change propagation.

In OTGen, Lerner and Habermann describe a system that addresses the issues of complex schema evolutions that require major structural changes to the database [Lerner and Habermann 1990]. Similar to ORION and O2, OTGen is assisted by a set of invariants that maintain schema consistency during the schema transformation process. The changes supported by OTGen deal with adding/deleting/renaming instance variables, changing types, adding/deleting superclass, and adding/deleting/renaming a class in the object schema. OTGen implements this using the schema modification method and does not support versions of schema. A very attractive feature of OTGen is that it modifies the structure and incorporates the existing data. Database administrators can specify the mapping between objects created in an old version and their corresponding representation in the new version of the schema using a declarative language. OTGen transforms the existing database to the new version automatically.
Zicari has proposed a set of schema change operations to be incorporated in O₂, an object-oriented database system and programming environment [Zicari 1991]. O₂ defines a distinction between types and objects, which ORION does not. O₂ has two distinct notions: classes whose instances are objects encapsulating data and behavior, while types have values as its instances. Every class is associated with a type, describing the structure of its instances. The object schema in O₂ is a set of classes related by inheritance links and / or composition links. The objective of this research is to define a minimal set of primitive operations that can be used to update the schema in O₂, while preserving the consistency of the schema. For this purpose two types of consistencies are identified. The schema is said to be structurally consistent if the class structure is a directed, acyclic graph (DAG) and if attribute and method name definitions, attribute and method scope rules, and attribute and method signatures are all compatible. The object-oriented database is behaviorally consistent if each method respects its signature and its code does not result in run-time errors. The important contribution of this research is that it addresses the issue of completely automating schema change process. The research emphasizes the importance of involving the designer in the process. It suggests providing a set of tools to the designer/user for the purpose of performing schema changes instead of automating the process entirely. A key issue not addressed in this research is the issue of propagating changes to the database. However, it is the only known research that identifies the provision of an advisory program that would determine at evolution time if the evolution were consistent with class and method dependencies.

Clamen describes a schema versioning method for schema evolution for distributed and centralized object-oriented database systems [Clamen 1994]. This research extends the concept of class versioning to all schema objects but explicitly addresses attribute evolution only. This
method allows for multiple versions of the instances, called facets, to persist simultaneously. The programmer is given the specifications on how to adapt the instances as a package following changes to the schema and uses a method similar to screening, but only when the programmer invokes the package. The focus in this article appears to be versioning and application management and not all schema changes are explicitly described.

Breche et al. further extend the schema change support for O2 by using views to simulate schema changes, in the GOODSTEP project [Breche et al. 1995]. An important issue addressed here is the support for modification of the database following schema changes, instance adaptation. Conversion functions are provided to the user, which take the old and new schema definitions as input and transform the objects of the database to conform to the new schema. This research adopts the lazy or delayed conversion similar to GEMStone and ORION. Views are treated as virtual schemas for simulating certain specific classes of schema changes. The designer has the option to test the change before materializing it.

The view mechanism in object-oriented databases has also been employed to create personal schemas from a base schema to assist in schema evolution [Ra and Rundensteiner 1997]. More specifically, by permitting “personal views”, changes requested by users are treated as changes to the personal schema instead of the database schema. Transparent Schema Evolution (TSE) system, a prototype built on this notion, now regenerates the personal view incorporating the requested change(s). Schema changes described include adding/deleting of object classes, adding/deleting "is-a" relationship edges between two classes, adding/deleting methods associated with objects, changing the domain of attributes, and renaming class objects. By defining virtual classes a new view that incorporates one or more of these changes is generated. The view generation is implemented using a set of algorithms, one for each type of
change. The propagation of changes is by filtering where the set of algorithms associated with each view determine how the data is to be mapped to that view.

TIGUKAT is an object-base management system that supports dynamic schema evolution [Peters and Özsu 1997]. An axiomatic model for dynamic schema evolution forms the foundation of TIGUKAT. Traditional approaches to schema evolution in object-oriented systems involve the definition of invariant properties and rules to enforce them, which makes the methodology system and model specific. An important contribution of this research is that it develops a formal model to which other object-oriented systems supporting schema evolution may be reduced to, especially for comparison purposes. The model is based on object type and uses the type-lattice graph to represent property inheritance, subtypes, and supertypes. The research does not address the propagation of the changes to the instances.

Examining the above research based on the steps involved in managing core schema evolution, the following observations can be made. All of the described research examines schema evolution in single standalone databases. First, for identifying possible changes to the schema based on the object model, some are very comprehensive in identifying changes while others focus on a specific subset of the changes. Second, for defining the consistent state of the schema, the preferred approach is to use a set of axioms to define the consistent state and further identify a set of rules to preserve the consistent state when changes are incorporated [Banerjee et al. 1987a]. Third, propagating the changes to the data instances is performed using conversion if multiple versions are not managed and is performed using screening/filtering if supporting multiple versions of the schema. The issue of dynamically managing schema evolution is addressed only in [Peters and Özsu 1997]. When using view mechanisms, the consistent state and its preservation are defined and controlled by algorithms associated with view generation.
and materialization. For this reason, these are often not explicit. The thrust in research using view mechanisms appears to be on version and application management than on core schema evolution. Lerner and Harberman highlight the issue of automating schema evolution by automatically generating the change schema after the changes are identified [Lerner and Habermann 1990]. This is also one of the few to comprehensively examine schema evolution including propagation of schema changes.

2.2 Core Schema Evolution and the Relational Data Model

The relational data model is simple when compared with the object-oriented data model and hence only a limited number of schema changes can be identified based on the relational model is limited. A majority of the research on schema evolution using the relational model examines temporal dimensions [EDBMS 1983], [Dadam and Teuhola 1987], [Clifford and Croker 1987], [McKenzie and Snodgrass 1990], [Ariav 1991], [Roddick 1991], [Scalas et al. 1993]. Other perspectives to tackle schema evolution in relational databases include extending SQL to support schema evolution [Roddick 1992], [Snodgrass 1995], automated schema transformation [Shneiderman and Thomas 1982], and using hybrid relations and special structures [Borgida and Williamson 1985], [Takahashi 1990]. A complete list is in table 2.

<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shneiderman and Thomas, 1982</td>
<td>Schema transformation</td>
</tr>
<tr>
<td>Borgida and Williamson, 1985</td>
<td>Exceptions handled by special data structures</td>
</tr>
<tr>
<td>Takahashi, 1990</td>
<td>Hybrid relations for exceptions</td>
</tr>
<tr>
<td>Evolutionary DBMS, 1983</td>
<td>Time stamping</td>
</tr>
<tr>
<td>Dadam and Teuhola, 1987</td>
<td>Time stamping and linked data structure</td>
</tr>
<tr>
<td>Clifford and Croker, 1987</td>
<td>Lifespan – time of validity/existence</td>
</tr>
<tr>
<td>McKenzie and Snodgrass, 1990</td>
<td>Temporal dimension – transaction time</td>
</tr>
<tr>
<td>Ariav, 1991</td>
<td>Temporally oriented data definitions</td>
</tr>
<tr>
<td>Roddick, 1992</td>
<td>Temporal dimension – valid and transaction time</td>
</tr>
<tr>
<td>Scalas, Cappelli, and De Castro, 1993</td>
<td>Temporal dimension – valid and transaction time</td>
</tr>
</tbody>
</table>

Table 2 – Summary of Core Schema Evolution using Relational Model
Shneiderman and Thomas examine a system and architecture for automatically converting relational databases following changes to its structure [Shneiderman and Thomas 1982]. This research classifies changes to the relational databases (transformations) based on three features. A transformation is *information preserving* if no information is lost. A transformation is *data dependent* if the data stored must be checked to determine if the transformation is consistent with the new structure. A transformation is *program dependent* if the application program must be checked to determine if the transformation is correct. Based on this taxonomy for transformations, a system for automatically transforming the database based on a given change is proposed. An important implication for automating such a process brought forth in this research is the need for the database administrator (DBA) to be involved in the conversion process especially when transformations do not meet consistency requirements. It further highlights the fact that such conversions are viable and systems that support semi-automated conversions provide DBAs with an increased flexibility by easing the conversion process.

Borgida and Williamson discuss a method that incorporates exceptional facts. These are facts that are consistent with the real world (which has changed), but do not conform to the current structure of the database (which has not changed) [Borgida and Williamson 1985]. The exception facts are stored in a separate logical file of records, each record holding a single exception fact. Query language is augmented with application programs that combine the logical file with the existing relational schema as and when required.

Takahashi describes the concept of hybrid relations to support schema evolution [Takahashi 1990]. The research is relatively simple in that it only describes the change due to addition of new attributes to an existing relation. This is achieved by storing the new attribute(s) of an existing relation separately. The new attributes are stored in the form of ordered triples.
consisting of \(<\text{tuple identifier, attribute name, value}>\). The tuple identifier is the set of values corresponding to the set of key attributes for this relation. At query time, the hybrid relation is realized virtually by combining the existing relation with its set of triples.

A commercial relational database management system, Evolutionary DBMS (EDBMS) has been developed by Information Research Associates [EDBMS 1983]. This system uses temporal concepts to support schema evolution, specifically the evolution of data definitions and supports dynamic restructuring of the schema. The approach taken is the management of binding time: the binding of definitions to their type-names and the binding of data types in programs that run against the database. The set of possible schema changes is not well defined in the description. Structure Manipulation Language (SML) is provided to implement schema changes and to coordinate changes between schema, data, and programs. EDBMS does not permanently convert old data to conform to the changed schema structure.

Dadam and Teuhola's Non-First-Normal-Form (NF\(^2\)) model examines schema evolution using temporal definitions [Dadam and Teuhola 1987]. The NF\(^2\) relational database permits the storage and manipulation of non-first-normal-form relations. The NF\(^2\) with temporal definitions handles time in the form of versions of data and schemas. Nine types of schema changes are addressed with the proposed method for implementing each using the storage structure described. Some complex schema changes addressed include nesting (Nest), undoing nesting operations (UnNest), Join, and Split operations which involve temporal restructuring of the schema.

Clifford and Croker introduce the concept of lifespan in their description of Historical Relational Data Model (HRDM) [Clifford and Croker 1987]. Each attribute value in HRDM is associated with a lifespan parameter that defines its period of existence. A group of such
attributes form a relation. A group of such relation schemes form the database schema. The lifespan attached to each attribute helps the model determine the period of time during which the attribute exists in the schema. This potentially permits evolution of the schema. Even though this research does not explicitly address the details of schema evolution, the lifespan concept provides an elegant method to address schema evolution.

Schema evolution results in changes to the database. Each change is an event at a point in time when the database records this change. This notion forms the basis for supporting schema evolution in temporal relational databases, by including support for transaction time in the relational model [McKenzie and Snodgrass 1990]. The basic data model assumed here supports four classes of relations: snapshot, rollback, historical, and temporal relations. Besides other changes, the relation may change its class (along with data and signature) as well. The relational algebra has been extended to support four schema change operations: adding a new relation, modifying an existing relation, deleting a relation, and renaming a relation. Implementation of these changes is described in depth using relational algebra. To ensure semantic correctness of the algebraic expressions with respect to the database schema and to maintain consistency between the state (data) and schema following changes, a system for semantic type checking is described. An interesting feature is the support for multi-command transactions in which more than one relation of the schema may be changed. A transaction is treated as an atomic update operation.

Ariav examines schema evolution using the Temporally Oriented Data Model (TODM) [Ariav 1991]. The central data structure proposed in this research is the data cube. The data cube is a temporal data construct in which time, objects, and attributes form the primary dimensions of the stored data. The stability of the data is the focus in this research and data is permanently
converted to conform to the most recent schema structure. An important contribution of this research is its emphasis and explicit addressing of managing applications when schema evolution occurs. Schema changes are classified as *temporally shallow* changes and *temporally deep* changes. The former includes changes that require relatively simple operations with minimal corrective steps to retrospectively apply the modified schema to existing data. The latter includes changes where retrospective application of the modified schema to existing data is difficult, and often requires substantial computations. Even though this research clearly outlines the requirements of schema evolution using the TODM and data definitions, it does not explain the details of performing the schema changes.

Roddick describes a model that incorporates temporal support into the metadata of the relational database [Roddick 1991]. Specifically, the addition of this temporal support has been investigated with reference to semantics of null values, its effect on integrity constraints and its impact on query languages. The temporal support provided includes both valid time and transaction time. Null values may pose serious problem in databases supporting schema evolution. Particularly, when new relations are added, all attributes of this relation may have null values. This would violate the entity integrity constraint and may violate referential constraints. The research suggests that the metadata-base (system catalog) must have temporal support and that query languages must query the metadata to identify the structure of the database during time instants or time intervals. This suggestion has been implemented by developing a query language SQL/SE to support schema evolution [Roddick 1992]. It employs the notion of complete relations. Every relation has an extension, its complete relation, with a set of attributes that is the union of all the attributes ever defined for that relation.
Scalas et. al. discuss another model for schema evolution in temporal relational databases [Scalas et al. 1993]. Changes are classified into two types: redefinition and revision. In redefinition, the schema after the change is completely independent of the schema before the change. Applying changes to the most recent schema “revises” the schema. The specific changes to the relational model discussed here are classified into two categories: those that deal with attribute level changes and those that deal with relation changes. The first category includes adding and dropping attributes, renaming attributes, and redefining attribute domains. The second consists of adding and dropping relations. Implementation details of redefinition and revision changes are described.

Methods for schema evolution using the relational model include temporal support for the database including temporal support for metadata, view mechanisms, and handling exceptions using special data structures. Both temporal support and providing view mechanisms focus more on the versioning support than on core schema evolution. Also a majority of the research on schema evolution using relational model do not explicitly consider propagation of schema changes to the database(s). Of particular relevance to schema evolution, these provide language support for schema evolution by extending and modifying relational algebra. The types of schema changes that can take place in a relational model and in relational databases are small (changes to attributes and to relations) compared with object data model and database. Providing temporal support (transaction and/or valid time) in databases is an excellent method for supporting schema versions. Extended SQL with added temporal support is essential for managing data as well as schema in these temporal databases [Snodgrass 1995]. View mechanisms primarily support versioning and are easily supported by relational models. These are addressed more in detail in the section dealing with version management. The works of
Takahashi and Borgida -Williamson do not address core schema evolution as we have defined it. They are included because they examine methods for accommodating changes without shutting down or reorganizing the database, supporting dynamic management of schema changes.

2.3 Core Schema Evolution and Conceptual Data Models

Solution methods for the management of core schema evolution have utilized conceptual data models such as the Entity Relationship Model (ERM) and NIAM. An important distinction between these models and the object-oriented and relational models is that the conceptual data models are not tied to any one type of database though ERM and NIAM models have been extensively used as a basis for deriving the design or schema of relational databases. Research using these models is summarized in this section and a summary list is in table 3.

<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for schema evolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Markowitz and Makowsky, 1987</td>
<td>ER-Model, uses ER-consistency and introduces ER-calculus</td>
</tr>
<tr>
<td>Andany, Leonard, and Palisser, 1991 (Farandole 2)</td>
<td>Semantic model similar to an ERM</td>
</tr>
<tr>
<td>Liu, Chang, and Chrysanthis, 1993,1994</td>
<td>ER-model, attribute changes</td>
</tr>
<tr>
<td>Roddick, Craske, and Richards, 1993</td>
<td>ER-model, attribute, relationship, &amp; entities</td>
</tr>
<tr>
<td>Ewald and Orlowska, 1993</td>
<td>NIAM model and fact-based evolution</td>
</tr>
</tbody>
</table>

Table 3: Summary of Schema Evolution using Conceptual Models

A methodology for implementing incremental and reversible schema change operations using the Entity Relationship (ER) model has been proposed by Markowitz and Makowsky [Markowitz and Makowsky 1987]. This research introduces the concept of ER-consistency and examines schema change operations in a relational database. It supports schema evolution by providing a method to incrementally restructure the relational database schema. A relational schema consisting of relational schemes, together with key and inclusion dependencies is said to be ER-consistent if an ER model can be used to represent the relational schema. The changes in an ER-consistent schema are the addition and deletion of relational schemes, and the modification...
of key and inclusion dependencies. The authors have developed an ER-calculus (an ER-oriented adaptation of tuple relational calculus) to represent database state mappings. Database state mappings define an ER-consistent database state. Propositions that ensure ER-consistency of the database states are also defined using ER-calculus. The ER model is treated as a directed graph and the graph is restricted to having three types of nodes representing entity classes, relationships, and attributes. The ER model (referred to as a role-free model) does not include cardinalities and assumes that relationships have no attributes. The schema restructuring operations are described as vertex-oriented changes to the ER model along with the database state mappings associated.

Farandole 2 is a prototype system that employs a semantic model that is similar to an ERM to manage schema evolution in object-oriented databases [Andany et al. 1991]. The model defines two types of classes, atomic and composite. Atomic classes are terminal classes (boolean, integer, strings, etc.) and every object in this class is identified by its value. Objects in composite classes are identified independent of their values and the value of each is made up of objects linked to it by roles. Roles are links that associate a composite class to another class that may be composite or atomic. The model permits specialization but not multiple-inheritance. The model is used to describe the entire schema or a context, which is a partial schema. A semantic context is an abstraction that permits regrouping of certain elements in the schema while excluding others. Any number of contexts may be defined on a database. A semantic context defined on a database is represented as a graph. Each node in this graph is a couple (n, C), where n is the node name, and C is the database class it is associated with. Each edge represents the role that links the classes associated with each node it is incident on. The evolutionary changes or transformations are of two types: those that are applied to the complete schema and those that are
applied to contexts of a schema. At the schema level a context version may be added or suppressed. At the context level, the changes are described in terms of changes to the context graph, addition and deletion of nodes/edges, modification of nodes/edges. Changes to names of nodes are not permitted. An interesting contribution of this research is that changes to the relational schema are described as changes to a semantic model and changes to a model-based graph. This research supports multiple versions of the schema and contexts.

Roddick et al. describe a taxonomy for schema evolution based on the ERM [Roddick et al. 1993]. This taxonomy classifies schema changes based on the ER model into 3 primary classes: changes to entity classes, to relationships, and to attributes. There is a separate taxonomy of schema changes to a relation model (not described) that defines relational algebraic operators for implementing changes. The proposed solution aims to provide a comprehensive schema evolution system for relational databases by combining the two.

Liu et al. address schema evolution changes by translating them into changes to an ERM [Liu et al.1993]. The schema changes explicitly described here are the changes in attributes (addition and removal of attributes), or changes to other objects that result in changes to attributes like specialization of entity types and changes to domain of attributes. A scheme for managing attribute changes is proposed using a concept similar to that used by Clamen [Clamen 1994]. Clamen employed this method to define relations between attributes of existing and evolved versions of a class. The key contribution of this work is the management of application programs during the schema evolution process.

Ewald and Orlowska describe a procedural approach to schema evolution using NIAM (Nijssen Information Analysis Methodology) model applied to relational databases [Ewald and Orlowska 1993]. The NIAM model is used to represent the real world using a fact-based
approach. Fact-based approach permits representation of a rich variety of constraints and hence perceived as being semantically expressive. Two types of schema changes are considered in the NIAM model, addition of a fact and deletion of a fact. The research provides interactive procedures to support the schema change operations. The interactive feature is essential as the NIAM model relies heavily on the expertise of the human designer. Interactive checking also preempts conflict problems associated with constraints and ensures that the semantics of the old schema is preserved.

Management of schema evolution has also been studied in hierarchical, network databases as an automated process for data transformation and restructuring. For completeness, we will briefly address these. WAND is a system that is based on a network model that permits dynamic restructuring of the database [Gerritsen and Morgan 1976]. The authors employ the concept of Generation Data Structures (GDS). In GDS, instead of one schema, generations of schema are maintained. Each logical data record is associated with a generation of schema that was in force at the time of definition of the record. Hence several generation of schema is permitted. This corresponds to the case where each change generates a new entire schema. Data is modified to conform to each new schema. A generalized method for restructuring a hierarchical database is described by [Navathe 1980]. In this model, the user needs to specify the source and target schemas. The model uses an internal representation, the schema diagram, which is represented as a graph. FORMAL is an application development system that permits automated transformation of data using an embedded Automatic Transformer [Shu 1987]. The user needs to specify the form of the output and the source of data via an interface provided.

Conceptual models have been used almost always in conjunction with relational databases. Andany et. al. use a graph-based representation of a specific conceptual model (that is
like ERM) for the purpose of describing the database and changes are described as addition/deletion of nodes and links in the graph. However, the focus of this work is on using views or contexts that are defined on the graph for the purpose of managing versions of schema in an object database. The issue of mapping changes to data is not explicitly addressed. Roddick et al. (1993) have proposed a mechanism that uses an ERM specifically for the purpose of managing schema evolution in relational databases. While a taxonomy is presented, implementation issues have not been addressed. Markowitz and Makowsky’s work is similarly restricted in that only changes to relational models (attribute and relation changes) are considered. Liu et al. examine attribute level changes only.

After examining research dealing with core schema evolution it is evident that a solution method that addresses this problem should include a framework based on which schema changes can be comprehensively examined. Almost all of the research on core schema evolution using object oriented models employ a formal framework supported by a set of rules or axioms that define the consistent state for the schema. The solution method must also include mechanisms to maintain the consistency of the database when changes are being applied. Operators or algorithms are used to make the changes to the schema and these enforce the consistency requirements defined in the framework. Further, changes to schema must be propagated to the data instances as well. It is also evident that research has not addressed schema evolution in multiple databases not to mention a heterogeneous set of databases.

3. Managing Schema Versions

Schema versioning is accommodated when the database allows the accessing of all data, both retrospectively and prospectively, through user definable version interfaces [Roddick 1995]. Management of schema versions includes, creating and maintaining multiple versions of the
schema following a sequence of changes, coercing data or mapping the existing data to the different versions, and identifying each schema version uniquely. A number of different strategies have been adopted for schema versioning. Versions of the entire schema may be maintained, or specific schema objects may be versioned and maintained (class or relation, type or domain). In addition, versions of the schema or of schema objects may be virtually created using views. Schema versioning strategies have been described using object-oriented systems, using relational database systems, and using conceptual models.

3.1 Management of Versions and Object-Oriented Data Model

Management of versions in object-oriented databases is accomplished at two different levels of the schema: versioning schema objects (class and type), or versioning the entire schema. In class or type versioning, the original schema is not modified. Instead a new version of the class or type is created. Using schema versioning, the entire schema is treated as a versioned object. A summary of these research methods is shown in table 4.

<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for schema versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Katz, Chang, and Bateja, 1986</td>
<td>Versioning support for design files</td>
</tr>
<tr>
<td>Skarra and Zdonik, 1986 (ENCORE)</td>
<td>Type versioning</td>
</tr>
<tr>
<td>Beech and Mahbod 1986</td>
<td>Object versioning</td>
</tr>
<tr>
<td>Bjornerstedt and Hulton, 1989 (AVANCE)</td>
<td>Object versioning extended to class objects.</td>
</tr>
<tr>
<td>Kim and Korth, 1990</td>
<td>Class versioning and view based support</td>
</tr>
<tr>
<td>Odberg, 1992</td>
<td>Class versioning</td>
</tr>
<tr>
<td>Monk and Sommerville, 1993 (CLOSQL)</td>
<td>Class versioning</td>
</tr>
<tr>
<td>Kim and Chou, 1988 (ORION)</td>
<td>Schema versioning using views</td>
</tr>
<tr>
<td>Bratsberg, 1992</td>
<td>Schema versioning using views</td>
</tr>
<tr>
<td>Clamen, 1994</td>
<td>Schema versioning</td>
</tr>
<tr>
<td>Ra and Rundensteiner, 1997</td>
<td>View based versioning</td>
</tr>
</tbody>
</table>

Table 4: Summary of Schema Versioning using Object-Oriented Model

Katz et al. describe a more general method applied to design files [Katz, Chang, and Bateja 1986]. They use a semantic object-oriented data model for representing how a complex
design database evolves over time. The design database is viewed as a large collection of objects. All objects together describe the product being designed. Certain objects are composed of other objects, called composite objects. Composition hierarchy represents this in the model. In addition, the model incorporates time in the form of version derivation hierarchies and time-varying configurations. The latter is used to correlate versions of component and composite objects. They describe methods to determine current version and support dynamic binding of configurations. It is one of the early works in version management. The concepts and constructs introduced using versioning of engineering design files (and databases) are now extended to versions of schema in evolving databases.

ENCORE provides a flexible mechanism called version set for dealing with version management of evolving types [Skarra and Zdonik 1986]. A version set is an ordered collection of all versions of a particular type. This set is initialized when the type is first defined and expands as new versions are created. The set is terminated when the type is deleted. Individual versions of that type may also be deleted in which case that version is made non-modifiable and non-instantiable. Versions with a set may be accessed randomly or sequentially using version numbers and one member of each version set is designated as the current version. An object is bound to a single version in the set. When an instance of a type is created, the user may define the version it belongs to. The default choice is the current version. Instances may be converted from one version to another of the same type (coercion). Properties, operations, and domain values defined for an object are defined by the specific versions of the type (and its supertypes) to which the object belongs. Affected program objects are dealt using a filter, the version set interface, which represents a union of all the properties, operations, and domain values defined by all versions of a type. Programs may interchangeably use the objects of a type created before
and after the change. The conversion method is performed when needed (late or lazy conversion). A key restriction of this system is that instances never change their type version. To overcome this problem Zdonik suggested a method by which an existing instance is provided with extra storage and a new interface enabling it to be a full-fledged instance of a new type-version [Zdonik 1990].

Beech and Mahbod discuss the functional requirements of objects and object-oriented models for supporting versions [Beech and Mahbod 1986]. Based on these requirements they define an object model for versioning where all information is represented by objects. The objects include system objects, type objects, function objects, literal objects. They define the relationship between objects and the versions of the object and the concept of generic versions. They describe mechanisms for implementing versioning operations that include creation of a versioned object both explicitly and implicitly, converting a non-versioned object to a versioned form, type checking and creating functions to ensure compatibility between versions of objects, and the definition of a context. The context provides a way to incorporate user-defined rules that govern the evaluation of functions applied to objects.

The AVANCE project develops a scheme for general versioning of objects [Björnerstedt and Britts 1988]. The object versioning support is extended to the versioning of class definitions as well. The system supported versioning mechanism is through the definition of Version Packets (VP). A VP is a sequence of identifiable points in history. A Version Set Packet (VSP) contains all the version packets of one generic object and maintains a version dependency graph to represent the derivation of versions. The VSP is a hidden construct maintained for bookkeeping purposes. Create, Derive, and Freeze are operations supported to manipulate VPs. VSP for an object is created when the first VP is defined on that object. AVANCE also supports
type versions. Similar to ENCORE, AVANCE adopts an exception handling approach similar to the version set interface. These exception handlers convert the data values to suit the version of the object specified by program requests. Versions are chronologically ordered and are accessible by system generated version numbers.

Kim and Korth describe a method for version management that utilizes object hierarchy [Kim and Korth 1990]. The unit of versioning here is the object class. The operations in managing versions include creating a new class version, creating special classes, modifying or deleting class versions, and accommodating versions of instances. A system class Object forms the root of the class hierarchy (DAG). For each class created, a generic class is defined as the immediate subclass of Object and serves as the superclass of all versions of that class. When a new class is created, its position on the DAG is determined by comparing it with all the class on the DAG under its generic class. Special classes that are created include dummy class (to accommodate instances of original class versions that are common to the original and new class) and equivalent class (for instances of a new class derived by renaming an instance variable of the original class). To manage creation of new instances two operations are defined. Adding a new instance to a class, deleting an instance from a class. In the latter case no changes to the DAG are needed as the instance is removed from the class version it belongs to. When an instance is added it could fit one version of the class, more than one version of the class, none of the versions of the class. In the first case the instance is added to the class version. In the second case, a new class version is created as the subclass of all the versions the instance could belong to, and the instance added to this new version. In the last case, a new version is defined, added to the appropriate position in the DAG and the instance is added to this new class.
A general approach for managing schema versions is proposed by Odberg [Odberg 1992]. This research employs a classical object-oriented data model based on a statically typed language C++. Two explicit constructs are defined in the model. *Type* is defined as the external representation of the object, i.e., externally accessible methods to which instances of this Type respond. Each type has an extent associated containing all the instances of that Type. Classes implement Types by providing methods defined on its Type(s) and on the attributes of that Type(s). The crux of this research is in explicitly distinguishing *Type* and *Class* (a Type is implemented by one or more classes). The unit for managing schema versions is the entire schema. Changes to Class are accomplished by introducing new Classes or Class versions and each version of the Class belongs to one schema version and all schema versions derived from this without modifying that version of the class. Type changes result in a new schema version and do not affect existing schema versions. A Union Set Type Version (USTV) is implicitly maintained for each Type and is a subtype of all versions of this type and includes a disjoint union of all methods defined on all versions of this type. The USTV provides a uniform interface that permits the modification of instances from any one version of a Type to another version of the same Type, transparently. Additional code modules are provided to perform these conversions.

Monk and Sommerville also adopt class versioning as the method for managing versions in their prototype system, CLOSQL [Monk and Sommerville 1993]. In a manner similar to ENCORE and AVANCE, CLOSQL maintains a set of versions of every class object that has evolved. Versions are numbered sequentially following the order in which created. However, CLOSQL differs in the manner in which it maps the data to the different versions. When a new attribute is added to a class in AVANCE, additional storage for the attribute is not allocated. The
implication is that this attribute can assume a fixed, read-only default value in the previous version(s) of that class. In CLOSQL, this is overcome by the use of special methods that convert instances from definition of one class version to the other. Unlike ENCORE, an instance is not bound to any particular class and its version identity is transparent to the user. The methods - backdate and update functions - are defined when a new version is created. Every version is hence linked to its previous version by the two functions. Conversions are performed when required once the version requested by the query is identified.

ORION adopts the schema versioning method using views [Kim and Chou 1988]. To maintain multiple versions of schema, a schema version could belong to six types: ancestor and descendent schema (derived from ancestor directly or indirectly), parent and child schema (directly derived from parent), current schema, and the creator schema. Creator schema is defined to identify the schema version under which a version instance is created. A set of seven rules defines the manner of derivation of versions and update capabilities of the versions. The schema is represented as a set of class objects where a class object is represented as a set of instances of several system-defined classes. A user-defined identifier identifies each version of the schema and objects belonging to a schema are identified by a combination of schema and object identifiers. View mechanism is used to manage versions and version instances that are tied to a specific version. To represent different schema versions, two structures are used. The schema version derivation hierarchy is represented as a tree rooted at the original schema. Each class object has an associated anchor instance that represents a sequence of schema versions. The first is the version under which the class object was created and each subsequent one is the version under which a copy of that class object was made due to changes to the class. A user interface consisting of a set of commands to manipulate schema versions is provided.
Bratsberg adopts a method for schema versioning based on views [Bratsberg 1992]. The objects in the schema, class objects, are treated as having two parts, the intent and the extent. Intents are evolved using views and a new version of the intent is created. Each class has a local extent, which represents the objects temporarily or permanently. A query to the local extents is defined to evolve extents. Depending upon the representation the query would include other extents that need to propagate objects to this extent. These query operations include identity-preserving selection, identity-preserving projection, object join, and set operators (union, intersect, and difference). Instances are associated with each new version of the extent. The instances are realized by defining a functional relationship between the instance variables of the original extent and those of the evolved extent version. This method does not support backward compatibility and null values can cause problems if the attribute did not exist before the evolution.

Clamen adopts schema versioning for managing versions following evolutionary changes to the schema [Clamen 1994]. Schema versions are numbered and identified uniquely. The object identifier identifies individual object in each schema version. Clamen proposes the provision of multiple interfaces, called facets, for accessing instances. Each facet encapsulates the state of the instance for each version. An instance with multiple facets is a disjoint union of the representation of each of the versions. Similar to backdate and update functions in CLOSQL, a set of functions help in the conversion of the instance from one version to another using the multiple facets. These functions relate the instance variables of the old version to those of the new version. The relationship between corresponding pairs of attributes in the old and new schema version determine how the functions compute the new values. These include shared (shared by both), independent (new attribute not affected by changes to the value of the old),
derived (new value derived from old), and dependent (new value may be affected by the value of the old, but cannot be computed from just the old values alone).

Ra and Rundensteiner, present a method called Transparent Schema Evolution (TSE) system that uses object-oriented view mechanism to define versions [Ra and Rundensteiner 1997]. When a change is to be made on a view schema, the system generates a new view that reflects the semantics of the change. The old view remains unchanged. When a change is requested from the initial schema, a view is created leaving the schema unchanged. Instead of copying every instance of the old schema to the new version, all object instances are shared by all versions of the schema. This is because all object instances are associated with a single global schema and each version is a view defined on this global schema. This approach also facilitates operations like merging two (or more) versions by avoiding consistency checking, removal of duplicates and other complications normally part of version merging.

Schema versioning methods using object-oriented data models have stemmed from engineering design databases. In engineering design databases, the designers introduce changes and the need to maintain different designs necessitated the need for managing multiple versions. Versions in object-oriented databases are identified by user specified names instead of system generated version identifiers. Version management methods in object-oriented databases range from maintaining versions of types and classes to maintaining versions of the entire schema. Hierarchies of versions are often represented by graphs and mapping between versions of a class, type, or schema is accomplished using mapping functions. A common thread that can be identified is the use of a generic version of a class, type, or schema. This is the union of all changes made to that class, type, or schema and individual versions are defined as views of this generic version. Beech and Mahbod (1986), Kim and Chou (1988), Bjornstedt and Hulton
(1989), Kim and Korth (1990) specify rules for deriving new versions, deleting existing versions, adding objects to versions, and deleting objects from existing versions. The others do not explicitly address these version derivation issues, but focus more on filters between versions, like Skarra and Zdonik (1986). Monk and Sommerville (1993), Bratsberg (1992), Clamen (1994), and Ra and Rundensteiner (1997) describe functions or algorithms to map the underlying data between versions.

### 3.2 Management of Versions and the Relational Data Model

Version management method using the relational data model is implemented by incorporating the time dimension in the relational model and versioning the schema based on time. A summarized list of the well-known research in this area is presented in table 5.

<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for schema versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EDBMS, 1983</td>
<td>Schema versions linked by definition trail. Data and programs identified by version number of corresponding schema version.</td>
</tr>
<tr>
<td>Dadam and Tuehola, 1987</td>
<td>Linked structures: current pool and history pool with links between. Time stamping to associate data and versions.</td>
</tr>
<tr>
<td>McKenzie and Snodgrass, 1990</td>
<td>Valid time and Transaction time support</td>
</tr>
<tr>
<td>Ariav, 1991</td>
<td>Data and Schema Cube</td>
</tr>
<tr>
<td>De Castro, Grandi, and Scalas 1997</td>
<td>Valid time and Transaction time support</td>
</tr>
</tbody>
</table>

**Table 5: Summary of Schema Versioning using Relational Model**

The Evolutionary DBMS maintains a definition trail for past versions of data definition [EDBMS 1983]. The concepts in EDBMS that support schema versioning include the Control Database (CDB) that serves as a library of data definitions, procedures, programs, and the relationships amongst them. All the elements of the CDB are versioned, starting from 0 and numbered in the increasing order. An object type in CDB is identified by a combination of type-name and a version number. The user schema is also versioned and identified using type-name
and version number pair. The Logical Access Manager (LAM) module in EDBMS implements
the concepts of logical data paths, including the run-time interpretation of the data structures and
fields in accordance with the relevant version of the schema. The data and compiled programs
are marked with the version number of the schema definition they correspond to.

The time-versioned Non-First Normal Form (NF²) model proposed by Dadam and
Tuehola handles time in the form of versions of schema and associated data [Dadam and Teuhola
1987]. An important observation is that, unlike other methods that incorporate temporal
dimensions into the relational model/database to manage versions, time is not included as a
dimension in the conceptual model that underlies NF2 database. The data storage in NF² is
divided into two pools, the current and the history pools. The main data structure residing in the
current pool uses pointer arrays to connect to all the multiple data elements (subtuples) within
the primary non-first-normal-form tuple. It thus has a network-like structure. The history pool
contains the records that represent the discarded values previously stored in the current pool.
Every element in the main data structure is linked to the discarded values in the history pool,
thereby providing the ability to reconstruct past versions of the tuples and subtuples. Data-driven
algorithms perform the realization of the versions. A version and its associated data are time
stamped and the algorithm uses the timestamps to match the schema with its corresponding data.
The algorithm first looks at the data involved, specifically, the timestamp of the data. Based on
the timestamp identified, the algorithm retrieves the corresponding version of the schema.

By incorporating transaction time in the temporal relational model, McKenzie and
Snodgrass describe a method for schema versioning [McKenzie and Snodgrass 1990]. Their
temporal model supports snapshot, rollback, historical, and temporal relations. Each change to
the schema is associated with this transaction time. A sequence of schema changes can therefore
be ordered by transaction time. Support for schema versioning in this research is allowing multiple schema definitions to be available and the rollback operator can be used to select the version required. A transaction number identifies each version of the schema. The algebraic language that supports database querying in this system also serves to manipulate the contents of the database and retrieve data corresponding to a schema version.

The data cube described by Ariav in his Temporally Oriented Data Model (TODM) captures time, objects, and attributes as the three primary dimensions [Ariav 1991]. The schema for the database is also represented along the same three dimensions, time, relations, and attributes. Thus changes to the schema may be sequenced by time and represented along with its data in the form of a schema cube, data cube pair supporting both schema and data versions.

Scalas, Cappelli, and De Castro examine a method for schema versioning in temporal relational databases that includes valid and transaction time [Scalas et al. 1993]. Four additional attributes - FROM and TO (for valid time), and IN and OUT (for transaction time) - are added to the schema of each relation. This ensures version support for the data. The structure of the meta-data catalogue is extended with the same four attributes and is defined for each relation in the catalogue. This ensures support for versions of the schema. A storage structure, the data pool, is defined to store tuples of relations that have the same structure. A single relation at the logical level may correspond to many data pools, as many as the number of incompatible versions of that relation produced by schema changes. Unique and unalterable identifiers REL_ID and ATT_ID identify a relation and an attribute. These combined with the temporal keys, identify versions of a relation and attributes.

Version management research using relational models has leveraged the use of temporal dimensions. Temporal relational databases that include both valid time and transaction time store
the metadata (data associated with database schema) in the same manner as they store data. Hence versions of schema (versions of meta-data) can be maintained. Transaction time supports rollback of historical relations, thereby maintaining versions of the schema. It is important to note that the metadata must be maintained as rollback relations and not as snapshot relations to completely achieve support for versioning. Temporal Structured Query Language (TSQL2) includes support for schema versioning [Snodgrass 1995]. If a relation has transaction time support then the ALTER statement defines a schema version for that relation. The schema itself becomes a set of transaction-time relations [Zanilo et. al. 1997]. The inclusion of valid time has been questioned as it defines how reality is modeled by the database [McKenzie and Snodgrass 1990]. However all schema versioning methods using temporal relational models include valid time. A key distinction can be observed between versioning methods using object-oriented models and those using relational models. The version identifier of relational model versions is always system generated. In object-oriented models the user defines the version identifier.

### 3.3 Management of Versions and Conceptual Data Models

In the large body of literature on schema versioning, very few have adopted conceptual models such as the ERM for schema versioning. Unlike object-oriented models conceptual models do not provide for the uniform treatment of the model and its underlying data and unlike relational models that have two basic constructs (attributes and relations) the conceptual models offer a number of different types of constructs. Versioning requires maintaining different versions of the model and versions of data associated with each. Based on these factors, conceptual model does not appear as an ideal tool for supporting schema versioning. A list of research employing conceptual models is given in table 6.
<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for schema versioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andany, Leonard, and Pallisser, 1991</td>
<td>Context or &quot;subschema&quot; versions based on a ER-like model.</td>
</tr>
<tr>
<td>Liu C. T., Chang, and Chrysanthis, 1993, 1994</td>
<td>View based versioning based on the ERM – explicitly address attribute evolution, domain evolution and specialization hierarchy changes only.</td>
</tr>
</tbody>
</table>

Table 6: Summary of Schema Versioning using Conceptual Models

Farandole 2 is one that does employ a data model, similar to an ERM, in implementing schema versioning [Andany et al. 1991]. This research adopts the notion of views, or contexts, to support schema versioning. In this model, a version is defined as a stable and coherent state of a context that the administrator or designer wishes to keep. A context is part of a schema. Hence versions of "subschemas" are managed. Generation of a new version is a decision of the administrator and not all schema modifications result in new versions. A version may be one of two types: a working version and a stable version. Changes are permitted in the former while the latter cannot be modified or updated. Conversion between versions is supported and at all times a designated default version of a context exists. Each version is associated with a generic context. Generic context has a root class that is specified by the user when a context is created. Semantically, the root class is the dominant class of the context. This root class cannot be modified or suppressed in any version of that context. The context is represented as a model graph in which nodes represent the classes and the links the roles between classes (see details in section 2.3). Changes to the context model need not necessarily result in a new version. If a new version is specified, then the model in the context is copied and changes made. The version information includes a system identifier, the identifier of its generic context, name, successors, predecessors, date, state (working/stable) and a list of nodes and edges in the version. A set of rules governs the manner in which a context may be modified.
Liu et al. maintain versions of database schema using views [Liu, Chang, and Chrysanthi
1993]. Even though the research does not explicitly address versioning mechanisms, it is
implicitly specified. The model of the entire schema is maintained, and only additions are
reflected. Deletions are stored in augmenting structures and a view is generated to represent
schema versions as a subset of the entire schema. The views of the schema versions are based on
the ERM representation of the schema.

The WAND model can support multiple generations of the database schema for a
network model [Gerritsen and Morgan 1976]. Of interest is the fact that the data corresponding
to each version is converted to conform to each version and stored.

Version management methods using conceptual models employ context-based or view-
based methods. Andany et al. define rules that govern the creation and modifications of versions
and use a graph structure to maintain version derivation. The unit of versioning is a context or a
partial schema and similar to versioning methods using object-models this research uses the
notion of a generic context. Liu et al. also use views of the ERM to maintain versions of schema,
but the version derivation process and rules are not discussed.

A comprehensive approach for version management in object-oriented databases would
have a combination of all three, viz., rules that define version derivation process, maintaining
version hierarchies based on a generic version, and mapping functions/algorithms between
versions to convert data instances. Each version may be generated physically, or generated
virtually using views with the latter being the more elegant and performance-friendly.
Introduction of temporal dimensions, specifically transaction time, appears to provide an elegant
method for managing relational database versions. Using views of conceptual models is one
method that employs conceptual models in versioning. Each view of the conceptual model needs
to be mapped to its corresponding view of the data. This clearly is more complex and would impose a considerable overhead especially in a heterogeneous database environment.

4. Managing Applications

Application programs that access a schema may be rendered obsolete when changes are made to the schema. Management of application programs is the task of providing mechanisms that help overcome this problem. Few have explicitly addressed this problem, though it has been implicitly addressed as part of schema evolution and/or schema versioning research. There are three predominant approaches to managing application programs. One method is by converting or modifying the applications so that they correspond to the new schema defined. The second approach is by maintaining versions of the schema and by providing a mechanism by which the appropriate data or version requested by the application can be identified. This mechanism may be part of the database system (extend query language support or conversion modules), filters between versions in the system, or provided as a library of functions for the applications. The third approach is the use of views. We now present a summary of research that deals with application management. These are classified based on the model used in the research.

4.1 Application Management and the Object-Oriented Data Model

A majority of research using object-oriented models has addressed the issue of application management using versions of schema. Maintaining versions provides a straightforward solution to application management. Applications continue to have access to a version of the schema that is compatible. However, applications must be told which version is compatible. The default version that is always the one available for access may not be the one that is compatible. We first describe research that employs this method. A summary of research managing applications using object-oriented models is given in table 7.
<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for Application Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Skarra and Zdonik, 1986 (ENCORE)</td>
<td>Filters between versions to convert data</td>
</tr>
<tr>
<td>Björnerstedt and Britts, 1988 (AVANCE)</td>
<td>Exception handlers to handle mismatch</td>
</tr>
<tr>
<td>Beech and Mahbod, 1988</td>
<td>User-defined functions to coerce instances. Support for user-defined functions provided using &quot;context&quot; structure. Applications invoke these functions</td>
</tr>
<tr>
<td>Monk and Sommerville, 1991 (CLOSQL)</td>
<td>Functions (update and backdate) to link each version with its predecessor and successor.</td>
</tr>
<tr>
<td>Bratsberg, 1992</td>
<td>Relate attributes of one version to that of another</td>
</tr>
<tr>
<td>Clamen, 1994</td>
<td>Relate attributes of one version to that of another</td>
</tr>
<tr>
<td>Osborn, 1989</td>
<td>Polymorphism and query equivalence</td>
</tr>
<tr>
<td>Ra and Rundensteiner, 1997</td>
<td>Views without changing original data.</td>
</tr>
</tbody>
</table>

**Table 7: Summary of Application Management using Object-Oriented Models**

ENCORE, the object-oriented prototype system for schema evolution supports application management using the version set interface [Skarra and Zdonik 1986]. The version set interface is an inclusive summary of all versions of a type. Every property and operation ever defined by a version of that type and every value ever declared valid for properties and operation parameters are represented in this interface. The significance of the version set interface is that it establishes a standard interface for all instances of that type, regardless of their version. Owing to the fact that a certain level of discrimination is used in defining the version set interface, errors may occur when a program attempts to access a particular version. To resolve this, error-handlers are introduced for every version. Error-handlers are programs (methods) that help smooth the differences between versions by converting instances from one version to another. They act as filters between versions. When a new version is defined by the removal of properties, domain values, or operations from an existing version, error-handlers are added to the new version to
convert missing values. If adding new properties, domain values, or operations creates the new version, error-handlers are added to the older version.

A similar approach is adopted in the AVANCE project as well [Björnerstedt and Hulton 1988, 1989]. Exception handlers are used to cope with mismatches between the version of the class expected by the query and that available. These serve the query with values appropriate to that version of the class the query expects.

CLOSQL defines update and backdate functions to link each new version of the class with its immediate predecessor thereby providing a chain of update and backdate functions that link all versions of a class [Monk and Sommerville 1993]. The update function converts instances from an older version to its successor. The backdate function converts from a newer version to its predecessor. Thus queries can access data corresponding to any version.

The methods proposed independently by Bratsberg and by Clamen for compatibility of applications to modified schema are largely similar in their notion [Bratsberg 1992], [Clamen 1994]. The idea is to identify a set of relations that relate instance variables of a class version to that of its predecessor and vice versa. Instead of specific functions that are defined in ENCORE, AVANCE, and CLOSQL, these provide a more general strategy to define the dependency relationships. Clamen defines that relationships between instance variables (attributes) of two versions may be classified as belonging to one of four types: shared, derived, dependent, and independent. In the first case, a synonym is created in the other version. For derived attributes, a function that determines the attribute value in terms of the attribute values in the other version is defined. For dependent attributes the function is based on attributes in both versions and independent attributes do not require anything. Bratsberg defines three types: dependent including derivable and non-derivable attributes, and shared attributes.
Polymorphism is defined as the ability of different classes of objects to respond to the same messages or carry out the same operations [Stefik and Bobrow 1986]. Osborn utilizes polymorphism and query equivalence to examine a solution method to manage applications [Osborn, 1989]. This research employs a standard object-oriented data model and defines an object-algebra on this model to study the extent to which polymorphism and query equivalence may be applied. Two types of schema changes are illustrated. The first is the conversion of a simple attribute to an aggregate class and the second deals with adding a cluster of subclasses to an aggregate class in the generalization hierarchy. The base model has two atomic classes (integers and variable length strings), an Aggregate class, and a class called Set. Aggregates have named components called attributes and the class Aggregate describes all the operations available on aggregates in the database. Set includes collections of objects from any class. The object algebra based data manipulation language provides for two kinds of equality testing. Two objects are identical if they are the same. Equivalence is defined differently for different classes of database objects. Operations on the classes defined using the algebra support polymorphism in that they can be applied to as many classes as possible and are shown to work for the two types of schema changes described.

Ling Liu et al. exploit polymorphism to help minimize re-programming of object-oriented applications [Liu et al. 1997]. In queries and applications, each application name used must be associated with a precise path expression to traverse the nested structure of the objects. If the schema structure changes, the path expression changes as well. The precise details of path expressions may be derived from the logical object structure of the corresponding schema. The notion in this research involves automated or semi-automated extraction of the path definition or the operational specification. The operational specifications are specified using a formalism -
propagation patterns. If these specifications are made available to be accessed by applications, then applications are transparent to structural schema changes. Polymorphism in these propagation patterns is utilized in that, with small refinements, these patterns could be re-used. The research also shows that re-programming can be minimized but not entirely avoided.

Transparent Schema Evolution (TSE) system makes use of the view mechanism to support schema evolution [Ra and Rundensteiner 1997]. TSE generates and realizes views of the original object-oriented database dynamically based on the changes specified. The advantage of using this method is that the original structure of the schema remains unchanged. Hence applications need not be changed and can continue to access a view that is compatible.

Support for applications in evolving object-oriented schemas is provided using filters. Filters are functions that sit between versions and map the data instances between versions. These functions may be user-defined (application specific) or generated semi-automatically (general dependency relationships) and take the form of exception handlers, backdate and update functions, and conversion algorithms. Application programs invoke them when the specific version requested by the application is identified. The onus of identifying the specific version may be on the applications themselves. Data corresponding to each version is converted when required following the lazy conversion scheme. Polymorphism appears to be an elegant method, however its support for all types of schema changes needs to be investigated.

4.2 Application Management and the Relational Data Model

In relational models, the inclusion of the time dimension (temporal relational databases) takes care of maintaining versions of schema and data. Support for applications to access historical data is provided in the form of temporal query languages like TSQL2 [Snodgrass et al. 1995]. Support for application management in relational databases with temporal support can be
provided by including a set of functions that map data to the different versions of the schema. Other approaches to application management in relational databases include support for automated modification and view-based versioning. A summary is presented in table 8.

Temporal models that include support for valid time and transaction time provide application support by supporting rollback operations and historical queries [McKenzie and Snodgrass 1990], [De Castro, Grandi, and Scalas 1997]. The data cube and schema cube model (TODD) proposed by Ariav, is also capable of providing application management support using historical queries and rollback functions.[Ariav 1991]. Ariav proposes the use of a single virtual schema with attached procedures for mapping the data between schema versions.

Roddick proposes an extension to SQL to support schema evolution in temporal databases [Roddick 1992]. This research supports valid-time and transaction-time, i.e., historical support and rollback support. Multiple versions of the schema may be maintained each identified by valid time and transaction time. The extension to SQL proposed, SQL/SE, for schema evolution permits the use of past or future schemas using [SCHEMA-FOR date-time | NOW | QUERY] operator. The operator [SCHEMA-AS-AT date-time | NOW | QUERY] provides rollback support. Both these operators are part of the extensions proposed.

The Evolutionary DBMS modifies the data to conform to the changed schema [EDBMS, 1983]. Even though it does not support versions, it contributes to application management through the provision of a specialized language and a utility program. The language, Structural Manipulation Language (SML) performs structural changes while coordinating the changes among schema, data, and programs. The utility module checks for inconsistencies between schema and applications and recommends remedial actions.
Table 8: Summary of Application Management using Relational Model

Temporal Structured Query Language (TSQL2) also provides support for legacy applications to access the old schema that was in effect when the applications were developed [Snodgrass 1995]. The operator SET SCHEMA DATE allows data written after or before a specific date to be viewed as per the schema in effect on that specific date. TSQL2 transforms data belonging to different schemas into the schema associated with a query. Some schema changes are not accommodated such as when a change causes a relation to be split into two (or more) relations.

Shneiderman and Thomas address the automated conversion of application programs as well in their architecture for automating the conversion of relational databases [Shneiderman and Thomas 1982]. This architecture supports conversion of existing data to conform to the changed structure. No versions are maintained and changes to the database structure may require changes to applications (program dependent changes). These changes are automatically made to the applications, and the administrator is informed of the changes. The administrator is responsible for accepting or rejecting the generated changes.
Temporal extensions to SQL and temporal query languages like TSQL2 are useful in managing applications using temporal relational models. Few like Roddick (1992) have explicitly addressed this issue. The method proposed by Shneiderman and Thomas (1980) does not support versioning but converts applications automatically based on changes to the schema. This implies all applications must be registered with the system so that they may be converted.

### 4.3 Application Management and Conceptual Data Models

To our knowledge few research methods employ conceptual models like the ERM to manage applications. A summary of these research methods is presented in table 9.

<table>
<thead>
<tr>
<th>Author(s) (and systems)</th>
<th>Method for Application Management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andany, Leonard, and Pallisser, 1991</td>
<td>Modifications eliminated for certain types of changes using views or contexts. Other types do need modifications</td>
</tr>
<tr>
<td>Liu C. T., Chang, and Chrysanthis, 1993, 1994</td>
<td>View based versions preserve the original model. Changes to database made by augmented storage and original structure and data is preserved.</td>
</tr>
</tbody>
</table>

**Table 9: Summary of Application Management using Conceptual Models**

Farandole 2 is one of two known works that employs a model similar to ERM in managing applications [Andany et al. 1991]. Farandole 2 employs the concept of contexts to maintain versions. Context (subschema) is like a view and is represented in the form of a data model. Application programs applied to context $V$ do not need modifications to $V'$ derived from $V$ except under one condition - the addition of a new class object to $V$ that resulted in $V'$.

Liu C.T. et al. use an extension of the ERM to manage schema evolution while ensuring that these changes are transparent to applications [Liu et al. 1993]. The changes described are changes to attributes of a class, domain changes, and specialization of entity classes. All of these require changes to attributes, the focus in this work. The ERM is treated as a graph and changes are implemented in terms of addition and deletion of nodes/edges to the graph. While addition of nodes / edges are reflected on the graph, deletions do not delete the affected object but treats
them as defunct. The user is required to specify the function that defines how deleted attributes are to be handled by the system. Views on the base model graph are used to represent versions and views do not affect the original graph. The structure of underlying database is modified based on the changes. New data is accommodated without changing the existing database structure. Instead complementary representations are used and these are related to existing objects in the database. Applications always have the schema (defined as a view of the base schema) that is compatible as well as the corresponding data.

Instead of changing the database structure if complementary representations could be used to capture new information, then applications will always work against the database. This method adopted by Liu et. al. appears to effective solve the application incompatibility problem. However, this method is suitable for a very limited set of schema changes, particularly those involving attributes only. In our opinion this is not a very suitable approach. The more suitable and complex method would require maintaining multiple versions of the schema (or subschema), a mapping between the versions and corresponding views of the database, and functions to coerce or map data to each of the database versions. No research has comprehensively addressed this aspect although Andany et al. have addressed a majority of these issues.

Application management is clearly an important part of schema evolution. It is also one area that has not received much explicit attention. The methods for managing applications are somewhat similar whether using object-oriented models or conceptual models. While both require versions of the model and the data, in object-oriented models these two may be one and the same. Application management in relational models is a far more challenging task as maintaining consistency between schema and applications is much more difficult.
4.4 Summary of Research

We have presented a summary of literature on schema evolution classified from two viewpoints: the activity managed (core schema evolution, management of versions, and application management) and the model used in to manage the activity (object-oriented, relational, and conceptual models like ERM).

Based on core schema evolution research using the object-oriented data model five basic notions can be identified: type modification, class modification, method modification, context or view-based modification, and schema modification. The first three address only a subset of all possible changes while the last two are more comprehensive in the set of schema changes addressed. Object-oriented models do not appear suitable for schema evolution in heterogeneous database environment, as object-oriented models tend to be database specific. Also important information such as constraints between classes remains hidden in the methods or behavior of the objects. Relational models appear similarly restricted, as the types of changes that can be made using a relational model are fairly small. Conceptual models appear well suited for addressing the problem of schema evolution in heterogeneous database environments.

Version management and application management are both very important and tie in very closely with schema evolution. In fact, it might appear that schema versioning preceded schema evolution. Many different methods for managing versions have been described, and amongst these, schema (class and type) versioning using views appears to be the predominantly used method. Filters (exception handlers, dependency relationships, algorithms, functions) are the most commonly used method for mapping data between versions. Versioning in relational models is achieved by using temporal dimensions such as the valid time and transaction time. To provide version support for the relational schema it is essential to represent the schema (meta
data) as rollback relations and not as snapshot relations. Versioning using conceptual models also adopt the view mechanism.

Application management has been acknowledged as being vital to schema evolution. However, it has not received much explicit attention. Using object-oriented models and using conceptual models, the filters employed between versions helps application management as well. In some cases, the version associated with the application is identified, either by the application or by the database system. In other cases, the application accesses the designated default version (usually the most recent one), and the filters map the data required by the applications. In relational databases with temporal support a requested version may be generated with the help of historical queries on the metadata. Relations could then be rolled back to realize the data associated. A summary list of all research in alphabetical order is in table 10.

<table>
<thead>
<tr>
<th>Authors</th>
<th>Model Used</th>
<th>Activities Addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andany, Leonard, &amp; Pallisser, 1991</td>
<td>ERM-like</td>
<td>Core schema evolution, versioning using views, application management</td>
</tr>
<tr>
<td>Ariav, 1991</td>
<td>Temporal Data Model</td>
<td>Core schema evolution, versioning, application management</td>
</tr>
<tr>
<td>Bertino, 1992</td>
<td>Object-Oriented</td>
<td>Core schema evolution using views, versioning.</td>
</tr>
<tr>
<td>Bjornerstedt &amp; Britts, 1988</td>
<td>Object-Oriented</td>
<td>Core schema evolution (class only), versioning, application management</td>
</tr>
<tr>
<td>Borgida and Williamson, 1989</td>
<td>Relational Model</td>
<td>Accommodating data without modifying schema structure.</td>
</tr>
<tr>
<td>Bratsberg, 1992</td>
<td>Object-Oriented</td>
<td>Core schema evolution, versioning, application management</td>
</tr>
<tr>
<td>Breche, Ferrandina, and Kuklok, 1995</td>
<td>Object-Oriented</td>
<td>Core schema evolution, versioning using views.</td>
</tr>
<tr>
<td>Clamen, 1994</td>
<td>Object-Oriented</td>
<td>Core schema evolution, versioning, application management</td>
</tr>
<tr>
<td>Clifford and Croker, 1987</td>
<td>Temporal Relational model</td>
<td>Historical queries - defines concept for versioning</td>
</tr>
<tr>
<td>Dadam and Teuhola, 1987</td>
<td>Non-first Normal Form</td>
<td>Time Versioning</td>
</tr>
</tbody>
</table>

Table 10: Summary of Schema Evolution Research
<table>
<thead>
<tr>
<th>Authors</th>
<th>Model Used</th>
<th>Activities Addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DeCastro, Capelli, and Scalas</td>
<td>Relational Model</td>
<td>Version and application management using valid and transaction time.</td>
</tr>
<tr>
<td>Ewald and Orlowska, 1993</td>
<td>NIAM</td>
<td>Procedural (fact-based) schema evolution</td>
</tr>
<tr>
<td>Katz, Chang, and Bateja, 1986</td>
<td>Object-oriented</td>
<td>Versioning in design files</td>
</tr>
<tr>
<td>Kim, H.T., and Korth, 1990</td>
<td>Object-Oriented</td>
<td>Versioning and application management</td>
</tr>
<tr>
<td>Lerner and Habermann, 1990</td>
<td>Object-Oriented</td>
<td>Core schema evolution with automatic database conversion</td>
</tr>
<tr>
<td>Lerner 2000</td>
<td>Object-oriented</td>
<td>Type evolution</td>
</tr>
<tr>
<td>Li and McLeod, 1988, 1991, 1994</td>
<td>Object-Oriented</td>
<td>Object Flavor Evolution, core schema evolution, application management</td>
</tr>
<tr>
<td>Liu, Chang, and Chysanthiss, 1993, 1994</td>
<td>ER Model</td>
<td>Core schema evolution (attributes only) and application management</td>
</tr>
<tr>
<td>Liu, Zicari, Hursch, and Lieberherr, 1997</td>
<td>Object-Oriented</td>
<td>Polymorphism dealing with application management</td>
</tr>
<tr>
<td>McKenzie and Snodgrass, 1990</td>
<td>Temporal Relational Model</td>
<td>Relational algebra for core schema evolution, valid and transaction time support for versioning, application management</td>
</tr>
<tr>
<td>Monk and Sommerville, 1993</td>
<td>Object-Oriented</td>
<td>Core schema evolution (class only), versioning, application management</td>
</tr>
<tr>
<td>Morsi, Navathe, and Shilling, 1994</td>
<td>Object-Oriented</td>
<td>Core schema evolution (method only) with consistency evaluation</td>
</tr>
<tr>
<td>Narayanaswamy and Rao, 1988</td>
<td>Object-Oriented</td>
<td>Specific problem of attribute evolution</td>
</tr>
<tr>
<td>Nguyen and Reiu, 1989</td>
<td>Object-Oriented</td>
<td>Core schema evolution of all schema objects</td>
</tr>
<tr>
<td>Odberg, 1992</td>
<td>Object-Oriented</td>
<td>Core schema evolution, versioning</td>
</tr>
<tr>
<td>Osborn, 1989</td>
<td>Object-Oriented</td>
<td>Core schema evolution explicitly addressing class and attribute evolution, application management</td>
</tr>
<tr>
<td>Penny and Stein, 1987</td>
<td>Object-Oriented</td>
<td>Core schema evolution (class only)</td>
</tr>
<tr>
<td>Peters and Öszu, 1995, 1997</td>
<td>Object-Oriented</td>
<td>Axiomatic model for core schema evolution</td>
</tr>
<tr>
<td>Ra and Rundensteiner, 1997</td>
<td>Object-Oriented</td>
<td>Core schema evolution, versioning using views, application management</td>
</tr>
<tr>
<td>Roddick, 1991, 1992</td>
<td>Temporal Relational Model</td>
<td>Core schema evolution, versioning using views, extending SQL for evolution</td>
</tr>
<tr>
<td>Roddick, Kraske, and Richards, 1993</td>
<td>ER Model</td>
<td>Core schema evolution taxonomy</td>
</tr>
<tr>
<td>Scalas, Cappelli, and DeCastro, 1993</td>
<td>Relational Model</td>
<td>Core schema evolution using valid and transaction time</td>
</tr>
</tbody>
</table>

Table 10: Summary of Schema Evolution Research (continued)
### Table 10: Summary of schema evolution research (concluded)

<table>
<thead>
<tr>
<th>Authors</th>
<th>Model Used</th>
<th>Activities Addressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shneiderman and Thomas, 1982</td>
<td>Relational Model</td>
<td>Automated transformation of databases</td>
</tr>
<tr>
<td>Skarra and Zdonik, 1986</td>
<td>Object-Oriented</td>
<td>Core schema evolution (type only), versioning, application management</td>
</tr>
<tr>
<td>Snodgrass et. al., 1995 (TSQL2)</td>
<td>Relational Model</td>
<td>Versioning support and support for application management for relations with transaction-time support.</td>
</tr>
<tr>
<td>Takahasi, 1990</td>
<td>Relational Model</td>
<td>Automated schema modification without affecting structure</td>
</tr>
<tr>
<td>Zicari, 1991</td>
<td>Object-Oriented</td>
<td>Proposal for core schema evolution</td>
</tr>
</tbody>
</table>

### 5. Schema Evolution and Heterogeneous Database Systems

After carefully examining existing literature in schema evolution it is evident that no research has addressed schema evolution in heterogeneous database systems. Managing schema evolution is even more challenging in heterogeneous database systems. Organizations today implement comprehensive information systems that have more than one data source and organizations maintain multiple databases. These databases may store different types of data, may be managed by different database management systems (DBMS), and may be distributed across geographically distributed locations. The schema of these underlying databases may also change. It is important to address schema evolution in such heterogeneous database systems.

Organizations today typically use an integrated collection of several databases creating a heterogeneous set. In this paper the term *heterogeneous database system* (HDS) describes a set of databases that store different types of data, are managed by different types of data management software systems (RDBMS, OODBMS, ORDBMS, and flat-files managed by 3GL), may be distributed across geographically dispersed locations, and data-associations (relationships) exist both within and across these multiple databases. Typically, a global or a federated schema is used to conceptually describe the world of reference of the HDS and
underlying databases. A change to any one database in this set could trigger changes to other databases in the HDS causing cascading changes because of the inter-relationships that exist. Each change can cause many more cascading changes in a HDS than in a single database. Also, whole individual databases may be added or deleted to/from the HDS triggering changes to the federated/global schema. Consider a large heterogeneous administrative database system for a university consisting of financial, research projects, employee, and student databases. When a space database (information on building(s) owned / leased / rented by the university, including office and classroom space) is added it triggers changes to the global schema as income/expenditure associated with each building needs to be tied in with financial data, research labs in buildings have to be linked to projects, and faculty/departments need to be tied in with buildings where they are housed. Given the number of changes and the cascading changes resulting from each, manually tracking and managing each change is difficult. Incomplete (and incorrect) tracking of changes would result in inconsistencies that affect the quality of data in the databases. It is necessary to automate the process to efficiently manage dynamic schema evolution in a HDS.

We examined a set of heterogeneous scientific databases used by researchers to study the effects of global climate change. Probes that are mounted on satellites, aircraft, and balloons measure data that is captured in this system. Over a one-year period, the number of new data files (corresponding to new information captured) increased from 38 to 82, an increase of more than one hundred percent. There were over 50 changes just due to additions and deletions of relationships. As the new data files were added to different databases in the heterogeneous system, new relationships between databases had to be defined and existing ones were sometimes deleted. In a different project involving the Student-Information-System, a set of
integrated databases used by the administrative office of a large university more than 20 tables were deleted or modified just over a two-week period. Several of these changes triggered cascading changes that affected other objects in other databases. Some of these changes were identified and incorporated immediately. Others were identified over a period of time prompted by users reporting inconsistencies in the data. Managing schema evolution in heterogeneous databases is a very complex task due to the inter-relationships that exist between the databases in the system. One solution to alleviate this problem is to automate the management of dynamic schema evolution. To do so, the following questions must be answered: what are the implications for automating schema evolution? Can it be completely automated?

Drawing from the literature the following tasks can be identified to manage core schema evolution in a heterogeneous database system. First, the schema must be represented using some data model. Given that the schema is a representation of a heterogeneous set of databases, object-oriented and relational models appear too database-specific. Semantic models appear to be very similar to object oriented models, the former have several advantages. Semantic models explicitly capture relationships between entity classes. They also provide the ability to explicitly model class constraints and attribute constraints. This information remains hidden in the definition of objects and methods in object oriented data models. Second, a comprehensive taxonomy of schema changes must be developed based on the model. The taxonomy would assist in understanding each change in terms of its implications so that appropriate operators may be defined to implement each. Third the consistent state of the schema must be defined based on the model and rules or axioms defined to identify inconsistencies and to preserve the consistent state. Four, comprehensively identifying all cascading changes in particularly complex and
important in a HDS and some automated support must be provided to perform this task. Finally, these changes must be propagated to the data.

From the literature it is evident that graphs and graph-theory has been widely applied to do the above (e.g. Orion [Banerjee et al. 1987], Frandole 2 [Andany et al. 1991], [Nguyen and Rieu 1989]). Using graphs also offers other advantages such as the ability to easily define operators (adding or deleting nodes or links) and graph-theoretic concepts (reachability) can be applied to trace and identify cascading changes. One approach would hence be to map the semantic model onto to a graph (a directed graph that is not acyclic) and then define the consistent state of the model and the schema based on the structure of the graph. This would also help define the operators for incorporating changes as graph-based operators that add or delete node(s) or links(s) to/from the graph. The graph representation of the semantic model schema would permit the use of graph-based algorithms utilizing “reachability” in graphs, to help identify parts of the schema that may be affected by a given change. Once the changes have been incorporated using the operators, the graph can be mapped back (reversed) to generate the semantic model schema. While this framework addresses the dynamic management of core schema evolution at the schema level it does not address propagation of schema changes. We propose the following architecture that uses metadata complete change propagation.

The three-tier architecture shown in figure 7 would incorporate the process and steps for managing dynamic schema evolution described in the previous paragraph. The change management layer consists of the current schema represented as a graph (called the semantic model graph). Schema changes are identified using the graph-based taxonomy in this layer. Graph based operators would incorporate changes to this graph and would be part of this layer as well. The axioms / rules defined based on the graph would help preserve the consistent state of
the graph (and associated schema). Once all the changes have been incorporated the modified graph can be mapped back to obtain the evolved schema. The propagation management layer consists of a metadata repository, a mapping dictionary (MD) and a Translator. The repository captures metadata on the semantic data model that represents the schema including metadata about cardinality constraints associated with relationships that cannot be captured in the graph. The MD maintains the mapping association between every semantic model (SM-graph) object and its corresponding database object as an ordered pair \(<O_x, D_x>\) where \(O_x\) refers to object \(x\) in the semantic model schema and \(D_x\) the corresponding database object. A database object might be distributed (parts of it spread over or copies of it duplicated or some combination of the two) across multiple databases. The MD includes a map for each database object and the corresponding set of one or more databases in the HDS in which this object is captured. The repository and the MD must be updated as each schema change is completed. A schema change could be initiated at the semantic model (the formal way) and completed at the data in the database. Alternately, the change could be initiated at the database level (e.g. adding a new attribute to a relation). In this case, the change needs to be mapped at the level of the semantic model and could be accomplished using the metadata and MD in the propagation management layer. The Translator, added for completeness, maintains information about the types of databases and DBMSs (relational, object, object-relational, flat-files etc.) and would generate editable (DDL) statements to implement schema changes in the databases. The database layer of this architecture consists of the set of databases in the heterogeneous data system.

To keep it simple, we have not included version and application management. The data in the set of underlying databases is converted to conform to the evolved schema (schema modification and not schema versioning). Some changes (the removal of a database object) will
result in loss of existing data. The DBA (and users) must be informed of and must approve of these changes before implementing each change. Further, in a heterogeneous database system the administrator's assistance would be required in deciding where to create a new database object (e.g. corresponding to the addition of an entity class), whether the object needs to be split or distributed across databases, and if so, the details of implementation. The DBA's assistance is also needed in deciding between alternative implementation choices (e.g. whether to extend a table in a relational database or create a new one to capture the details of a new relationship). The answer to many of these questions are fairly simple and intelligent assumptions may be made that would circumvent the need to interact with an administrator. There are some issues however, that need input from the administrator or expert user(s).

Figure 7: Architecture for Automating Schema Evolution in a HDS

Using the architecture shown it is possible to automate the management of dynamic schema evolution in databases. However, based on the extensive involvement of the database
administrator and/or expert users, it would be impossible to completely automate the process. At this stage, management of dynamic schema evolution in HDS can at best be semi-automated.

6. Conclusion and Research Directions

A large body of literature on schema evolution exists today reflecting the interest and extensive work completed in this area. However, the management of schema evolution in a heterogeneous database system has not received any attention from the research community. The primary objective in this paper was to present a comprehensive summary of the literature, pointing out the different activities and data models used by researchers. The second objective here was to draw attention to the importance of dynamically managing schema evolution in a heterogeneous set of databases. While literature has suggested automating the management of schema evolution, little information is available on implications for automating it and the extent of automation possible. The third purpose of this research was to suggest a mechanism for automating dynamic schema evolution in a heterogeneous database system and using this to examine the implications for automating it. It is hoped that this would serve as a foundation for further research on this very interesting and challenging topic.

Managing schema evolution has several interesting and challenging applications. One of these is its application to heterogeneous database systems that has been discussed in the earlier section. There are several others and the following are some (not exhaustive by any measure) areas where managing schema evolution is challenging and important.

Data warehouses are built using existing databases. Data warehouses are therefore susceptible to schema changes. Warehouses also have a large number of application programs that extract, transform, and load data from the operational and legacy data sources into the warehouse. Schema changes to one or more of these source databases would seriously and
adversely impact these ETL applications. Managing schema evolution dynamically in warehouses would help identify affected applications, propagate the changes to these applications, and with the help of the warehouse administrator, help maintain data consistency between the warehouse and its source databases.

Organizations capture business knowledge as knowledge objects. Each such object is a packet of value-added information that is self-contained and preserves the content and context from its original business setting for reuse in other settings. The structure of each object is determined based on the intended use/application of the knowledge stored and limits it to this purpose. The knowledge repository consists of several such objects and each of these is related to several other objects in the repository. The inter-relationships capture the fact that decisions, actions, and/or lessons captured in a knowledge object is based on or forms the basis of decisions, actions, or lessons captured in several other knowledge objects. These objects change, are removed, or new objects added over time where changes are driven by changes to the content and by changing user requirements. The framework described here is being applied to dynamically manage changes in knowledge repositories.

The advent of mobile access to information using a variety of different devices has strong implications for schema evolution. Organizations are now faced with the need to support access to organizational information from a variety of devices ranging from desktops, laptops, cellular phones, pagers, and palm-top computers and PDAs. Each type of device requires data of different granularity (in terms of details). While phones and pagers (and some palm-tops) can accept data in summarized forms others such as laptops and desktops can accept more detailed information. Moreover, these devices with their wireless capabilities may request for data from different and varying points on the organizational network. One solution that is being examined
is to create databases at the different nodes in the network. Each database stores the often-accessed organizational information in different forms – highly summarized to very detailed and the range in-between. Each database must be capable of dynamically changing its structure to store the range of information depending upon the type of data access and device it is required to support. The access type and device may change depending upon the time of the day and the set of users whose locations could change during the day. The schema of each database must be dynamically changed to accommodate the users and their data needs.

REFERENCES


BATORY, D. S. and KIM, W. Modeling concepts for VLSI CAD objects, *ACM Transactions on Database Systems*. 10, 3 (September 1985), 322-346


CHEN, P. The Entity Relationship Model - Toward a unified view of data. *ACM Trans. on Database Systems. 1*, 1 (March 1976), 9-36


CLIFFORD, J. and CROKER, A. A Historical Relational Data Model (HRDM) and algebra based on Lifespans. *Proceedings of 3rd IEEE Int. Conference on Data Engineering*, Los Angeles CA (1987), 528-537


GIBBS, S. J. and TSICHRITZIS, D. A Data modeling approach for office information systems. ACM Transactions on Office Information Systems, 1, 4, (October 1984), 299-319.


KATZ, R.H., CHANG, E. and BATEJA, R. Version modeling concepts for CAD databases. SIGMOD Record, 15, 2 (May 1986), 379-386.


LI, Q. and MCLEOD, D. Conceptual database evolution through learning in object databases, IEEE Transactions on Knowledge and Data Engineering, 6, 2 (April 1994), 205-224.

LIU, L., ZICARI, R., HURSCH, W., and LIEBERHERR, K. J. The role of polymorphic reuse mechanisms in schema evolution in an object-oriented database. *IEEE Transactions on Knowledge and Data Engineering*. 9, 1 (1997), 50-67


MONK, S. and SOMMERVILLE, I. Schema evolution in OODBs using class versioning. *SIGMOD Record*. 22, 3 (September 1993), 16-22


OSBORN, S.L. The role of polymorphism in schema evolution in an object-oriented database. *IEEE Transactions of Knowledge and Data Engineering*. 1, 3 (September 1989), 310-317


RA, Y. G. and RUNDENSTIENER, E. A. A Transparent Schema-Evolution system based on object-oriented view technology. *IEEE Transactions of Knowledge and Data Engineering*. 9, 4 (July/August 1997), 600-623


RODDICK, J. F - SQL/SE - A query language extension for databases supporting schema evolution. *SIGMOD Record*. 21, 3 (September 1992), 10-16


