

Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships

Chien-Tsai Liu, Panos K. Chrysanthis*, Shi-Kuo Chang

Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260

Abstract. *A flexible database system needs to support changes to its schema in order to facilitate the requirements of new applications and to support interoperability within a multidatabase system. In this paper, we present an approach to schema evolution through changes to the Entity-Relationship (ER) schema of a database. We enhance the graphical constructs used in ER diagrams, and develop EVER, an EVolutionary ER diagram for specifying the derivation relationships between schema versions, relationships among attributes, and the conditions for maintaining consistent views of programs. Algorithms are presented for mapping the EVER diagram into the underlying database and constructing database views for schema versions. Through the reconstruction of views after database reorganization, changes to an ER diagram can be made transparent to the application programs while all objects in the database remain accessible to the application programs.*

1 Introduction

As the reality of interest, usually captured by a database, changes over time, there is a need to be able to reflect these changes in the database. In this way, the requirements of new database applications can be facilitated. However, a database stores information for a long time and, in general, it is neither easy nor practical to re-structure a large database. Furthermore, it is not easy nor practical to modify complex application programs such as those found in database systems [15]. Thus, there is a need to continue supporting existing application programs, providing access to objects created under previous or new database schemas. Similarly, new applications should be able to access existing objects.

Supporting consistent access to objects created under different schemas is a requirement also in the context of multidatabase systems [10, 16, 22]. Here, component database schemas, possibly corresponding to different data models, need to evolve into a common, integrated, multidatabase schema that supports efficient and transparent data sharing among the component databases. Since a multidatabase system does not support complete integration, each component

* This work is partially supported by the N.S.F. under grant IRI-9210588.

database system continues to operate in an independent fashion providing access to its database through its existing local database schema.

Various approaches to the problem of changing database schemas and maintaining consistency between instances created under different schemas have been proposed, particularly in the context of object-oriented databases (OODBs) [3, 4, 8, 2, 19, 23, 24]. In this paper, we present a different way to support schema evolution, one based on the (*Extended*) *Entity-Relationship* (ER/EER) approach for data modeling [6, 21]. We also describe the implementation of the ER-based databases and their schema evolution in the context of the relational and OO data models. We chose to examine the semantics of changes in the context of the ER model for two reasons. Firstly, the ER approach provides a graphic oriented representation of a database schema, namely ER diagrams, which are closer to the designer's perception of data, rather than to the logical database schema which describes how data are stored in the database. Secondly, we wanted to avoid defining yet another object-oriented model that would allow for more types of relationships in order to support schema evolution. Since the ER model supports many types of relationships, it has the potential of becoming object-oriented [7, 13], and hence, effectively supporting the mapping of an ER schema into any object-oriented one.

In our approach, a new schema is derived with the help of an *Evolutionary ER diagram* (*EVER diagram*). EVER diagrams are ER diagrams enhanced with schema evolution graphical constructs that provide for the specification of the derivation relationships between schema versions, the relationships among attributes, and the conditions for maintaining consistent views of application programs. Toward this end, we derived a classification of attribute relationships between schema versions. EVER diagrams are transformed into an internal database representation called a *version derivation graph* (*VDG*) which is subsequently mapped into the structures of the underlying database. With respect to the underlying database, each schema version is expressed as a (database) *view*. The view constitutes the actual interface to the application programs for accessing the objects in the database. Thus, views are reconstructed after a database reorganization so that schema changes are made transparent to the application programs while all objects in the database remain accessible to all the programs.

In the next section we survey the various object-oriented approaches to schema evolution some of which were proposed to support schema integration. In Section 3, we formally analyze and classify the relationships of attributes between the schemas before and after a change and discuss the issues in maintaining a consistent database manipulated through different schema versions. EVER diagrams are first introduced in Section 4 whereas their transformation into an underlying database is discussed in Sections 5 and 6.

2 Related Work

Schema evolution has mainly been investigated as an aspect of OODBs. Broadly, approaches to schema evolution can be classified into *schema modification*, *schema*

versioning and *schema derivation*, based on the *external representation* of the structure of the objects in the database (object schema) to application programs, and the *internal representation* of the objects in the underlying database.

Schema modification approaches always support a single schema and a single internal representation for each object [2, 5, 24]. Hence, all objects must be converted to conform to the new schema. Because of this, the schema modification approach does not support the transparency of change for the existing application programs. The application programs that use the old schema may need to be modified.

Schema versioning approaches support multiple schemas and multiple internal object representations for an object [1, 19]. The instantiation of objects to a schema version is performed at the time of the creation of the objects. In this approach, the objects belonging to a version of a schema always stays in that version. Thus, if the schema of the objects is subsequently augmented, it would not be possible for the objects to be updated by the programs associated with a later version without *loss of information*. In an old version, an augmented attribute may have insufficient storage.

Schema derivation approaches [3, 4, 8, 23] support multiple schemas for an object and a common internal object representation. Irrespective of whether objects are created under different schema versions, they are converted to a common representation. The instantiation of objects to a schema version is performed at run-time. That is, the objects are presented to the programs as views on objects in the underlying database. Although existing derivation approaches allow any schema version of an object to evolve, it is not clear how object consistency can be specified and maintained across schema versions derived from different paths.

Our approach belongs to the family of schema derivation approaches that supports linear schema evolution (i.e., only the most recent schema can be evolved). By considering discontinuities of attribute relationships in the evolution history, our approach effectively resolves the object consistency problem mentioned above.

3 An ER Model Schema Evolution

When a schema is changed, a new version of the schema (or new schema) is then created. In the context of ER model, we use *schema* to refer to the description of an entity type or a relationship type and *object* to refer to an instance of an entity or a relationship type. Each schema version is the interface for programs to access the database. Our approach supports linear evolution of database schema. That is, a designer can only make changes to a new schema. The old schema versions are mainly used for supporting the existing programs.

In this section, we analyze the attribute relationships between two schema versions, and explicitly express these relationships in terms of functions. Then we discuss the problem of *discontinuity* in specifying attribute relationships for any two schema versions and the way our approach deals with this problem. Finally, we present rules for maintenance of a consistent database.

A. Analysis of Attributes in Different Schema-Versions

When a schema evolves, the relationships between the attributes of the old and the new schema capture the semantics of installed changes. These relationships provide the crucial information for maintaining object consistency and reorganization of the objects in the underlying database. In addition to the attribute values [8], we classify the attributes between two schema versions based on the relationships of their names and their domains.

- Common attributes: An attribute is said to be *common* to the two schemas, if the name and domain of the attribute in the two schemas is identical.
- Domain-changed attributes: An attribute is said to be *domain-changed* if its name in the two schemas is the same but its domain is different.
- Renamed attributes: An attribute is said to be *renamed* if the attribute in the two schemas has different names but exactly same domains.
- Resumed attributes: An attribute is said to be *resumed* if the attribute was deleted from the old schema but it is added back to the new schema.
- Derived attributes: An attribute is said to be *derived* if the value of the attribute can be derived from the values of other attributes not necessarily of the same schema version.
- Dependent attributes: An attribute, let us say B , is said to be *dependent* if the value of the attribute is affected by changes to the values of attributes in other schema versions, let us say $\{A_1, A_2, \dots, A_k\}$, but the value of the dependent attribute cannot be *derived* from the values of the same attributes $\{A_1, A_2, \dots, A_k\}$.
- Independent attributes: An attribute is said to be *independent* if its value neither affects, nor is affected by, the values of other attributes. If the attribute is an attribute of the new schema, it is called *new* attribute. On the other hand, if the attribute is an attribute of the old schema, it is called an *eliminated* attribute.

Derived and dependent attributes are further distinguished into four groups depending on where they are defined. If $\{A_1, A_2, \dots, A_k\}$ are attributes of the old schema, and B is an attribute of the new schema, then B is classified as *forward*. If $\{A_1, A_2, \dots, A_k\}$ are attributes of the new schema, and B is an attribute of the old schema, then B is classified as *reverse*. If $\{A_1, A_2, \dots, A_k\}$ can be attributes in the new schema or old schemas, and B is an attribute of the new schema, then B is classified as *forward complementary*. However, if B is an attribute of the old schema, then B is a *reverse complementary*.

Let us illustrate the attribute relationships using an example shown in Table 1. In this example, the old schema of Car database consists of attributes RegNo (Registration Number), Model, Color, WarrantyBegins (the year that the warranty of the car is initiated), EngineType and Fuel (*leaded* or *unleaded*). After evolution, the new schema contains attributes VIN (Vehicle Identification Number), Model, Color, WarrantyExpires (the year the warranty is expected to expire), WarrantyExtension (the number of years that the warranty has been extended), and MPG (mileage per gallon). From the table, it can be seen that attribute Model is *common* to both schemas, whereas attribute VIN in the new schema is a *re-*

The old schema	The new schema
<i>RegNo</i> : <i>string</i> [20]	<i>VIN</i> : <i>string</i> [20]
<i>Model</i> : <i>string</i> [20]	<i>Model</i> : <i>string</i> [20]
<i>Color</i> : <i>integer</i> [1..256]	<i>Color</i> : <i>string</i> [10]
<i>WarrantyBegins</i> : <i>integer</i> [1900..1999]	<i>WarrantyExpires</i> : <i>integer</i> [1900..2025]
	<i>WarrantyExtension</i> : <i>integer</i> [0..10]
<i>EngineType</i> : <i>string</i> [20]	
	<i>MPG</i> : <i>integer</i> [1..100]
<i>Fuel</i> : <i>char</i> [<i>leaded</i> , <i>unleaded</i>]	

Table 1. The evolution in a database schema for cars

named attribute corresponding to attribute *RegNo*. Also the domain of attribute *Color* in the new schema is different from that of *Color* in the old schema; thus, *Color* in the new schema is a *domain-changed* attribute. Attribute *MPG* in the new schema is dependent on attribute *EngineType* in the old one because when the type of a car engine is changed, the mileage of the car may also need to be changed. Thus, *MPG* is *forward dependent* on attribute *EngineType*. In the same way, when the mileage of the car is changed, the type of the car engine may need to be changed, too. Thus, *EngineType* is *reverse dependent* on attribute *MPG*. More interesting, the derivation of values of attribute *WarrantyExpires* in the new schema involves attribute *WarrantyExtension* in the new schema and attribute *WarrantyBegins* in the old schema. Let us assume that the default warranty period is one year. The attribute *WarrantyExpires* can be expressed as follows.

$$WarrantyExpires = WarrantyBegins + 1 + WarrantyExtension.$$

Thus, attribute *WarrantyExpires* now becomes a *forward complementary derived* attribute. On the other hand, if attribute *WarrantyExtension* and *WarrantyExpires* have been in the old schema, and attribute *WarrantyBegins* in the new schema, then attribute *WarrantyExpires* would be a *reverse complementary derived* attribute. Finally, attribute *Fuel* is an *eliminated* attribute in the old schema. On the other hand, if the attribute *Fuel* had been in the new schema, but not appear in the old one, then it would be a *new* attribute.

B. Specification of Attribute Relationships

The relationships of each attribute group can be expressed with a help of functions. Our framework uses four kinds of functions:

Identity function. If attributes *a* and *b* are always identical, their relationship can be represented by using an *identity function* (*I*): $a = I(b)$.

Derivation function. If an attribute *a* can be derived from b_1, b_2, \dots, b_k , the relationship of *a* to attributes b_1, b_2, \dots, b_k can be represented by a *derivation function* (*f*): $a = f(b_1, b_2, \dots, b_k)$.

Prompt function. If an attribute a depends on attributes b_1, b_2, \dots, b_k but it cannot be derived solely from b_1, b_2, \dots, b_k (e.g., it may need additional information), the relationship of a to attributes b_1, b_2, \dots, b_k can be represented by using a *prompt function* (Ψ): $a = \Psi(b_1, b_2, \dots, b_k, \Phi)$, where Φ represents the additional information. Φ is possibly an interactive query against the whole database. For example, if a car's EngineType is updated through the old schema, then the car's mileage needs to be updated (see Table 1). The car's MPG can be acquired by either prompting a user or extracting it from the information in the database, e.g. *Engine-Mileage cross reference table*. At this point it is interesting to point out the difference between derived and prompt functions. A derivation function captures local object consistency requirements, whereas a prompt function captures database consistency requirements, and as such, it is executed against the whole database (globally) and not within the object (locally).

Default function. If the value of an attribute a in an object is unspecified but the value is required by an application program, then the value can be acquired by invoking a *default function* (*default*). By assigning a default value to an unspecified attribute value, the need of the application programs associated with different schema versions can be resolved.

Attribute Group	Associated function	
	Old schema	New schema
common	reverse identity function	forward identity function
domain changed	reverse derivation function or reverse prompt function	forward derivation function or forward prompt function
resumed	reverse identity function	forward identity function
forward derived		forward derivation function
reverse derived	reverse derivation function	
forward-complementary derived		forward-complementary derivation function
reverse-complementary derived	reverse-complementary derivation function	
forward dependent		forward prompt function
reverse dependent	reverse prompt function	
forward-complementary dependent		forward-complementary prompt function
reverse-complementary dependent	reverse-complementary prompt function	
new		default function
eliminated	default function	

Table 2. The association of functions with attributes

These functions can be used to ensure object consistency for update and retrieval operations. For example, suppose three consecutive schema versions V_{i-1} , V_i and V_{i+1} along a schema evolution course. When an attribute a in V_i is updated, the new value of a is propagated to the attributes of V_{i-1} that are derived/depended on a using reverse or complementary reverse functions. At the same time, the new value of a is propagated to the derived/depended attributes in V_{i+1} using forward or complementary forward functions. The possible associations of functions with attributes are summarized in Table 2.

C. Discontinuity of Attribute Relationships

Although the attribute relationships between attributes of two arbitrary schema versions can be computed transitively using the specified functions, there may exist an attribute relationship between two schema versions that cannot be computed from the specified relationships between consecutive schema versions. We will elaborate on this problem using the example shown in Table 3.

V_1	V_2	V_3
$VIN : string[20]$	$VIN : string[20]$	$VIN : string[20]$
	$MPG : integer[1..100]$	$MPG : string[3]$
$WarrantyBegins : integer$		$WarrantyExpires : integer$
$EngineType : string[20]$		

Table 3. The problem in representing attribute relationships

Let V_1 , V_2 and V_3 be three consecutive schema versions of a student database. The relationship of attribute *WarrantyExpires* in V_3 to attribute *WarrantyBegins* in V_1 cannot be correctly captured, because attribute *WarrantyExpires* in V_3 is new with respect to V_2 and attribute *WarrantyBegins* in V_1 was eliminated with respect to V_2 leading to the conclusion that these attributes are independent. However, as we saw above, attribute *WarrantyExpires* in V_3 can be derived from attribute *WarrantyBegins* in V_1 . This phenomenon is due to the discontinuity in the evolution history of the attribute.

Such a discontinuity would have not occurred, if the designer had first “resumed” the previously eliminated attribute, and then followed the regular change procedures. In this example, the designer must first interpose an intermediate schema version $V'_2(VIN, MPG, WarrantyBegins)$, between V_2 and V_3 . In V'_2 , *WarrantyBegins* is a *resumed* attribute from V_1 . We call V'_2 a *resumed* schema version. The relationship of the resumed attribute in the two schema versions can be expressed similar to a common attribute in terms of an identity function.

Thus, in our approach, a schema evolves from the resumed schema corresponding to the latest schema version. The resumed schema version cannot be seen by application programs and is constructed during schema evolution by

combining all the attributes of the latest schema version with all the eliminated attributes of the previous schema versions. In this way, we re-establish discontinued attribute relationships and therefore, we can reconstruct the relationships between any two schema versions.

D. Maintaining Database Consistency Across Schema-Versions

Informally, a database is said to be consistent if two observers who view the database through different schema versions see an object in ways that agree with each other. In our framework based on ER schema evolution, we completely avoid the modification of application programs, by ensuring a consistent database along three dimensions: *object consistency*, *key consistency*, and *invariant program views*.

Object Consistency. As discussed in the previous section, the maintenance of object consistency can be accomplished through the specified functions. Whenever an attribute value of an object is updated, those attributes that depend on it must be updated by using the specified functions.

Key Consistency. The key consistency specifies *the uniqueness of the objects across the old and new schemas*. That is, each object, irrespective of whether it is created by the old or new schema, must be uniquely identified by using the values of the key attributes defined in the old and new schema. The maintenance of key consistency cannot be performed by the integrity constraints alone because the key attribute may be different in the different schema versions. Therefore, in our approach, we enforce the following condition when a designer changes the key attribute: *the mapping of the key attributes between the new and old schemas must be one-to-one*.

Invariant Program Views. The invariant program views specify *the semantics of a database for the programs associated with a particular schema version*. However, the evolved database may not preserve the interpretation made by the programs associated with the previous schema versions. Since the views of programs to a database are application dependent, in our framework, we provide facilities to allow a designer to specify the conditions under which programs retain consistent view of the evolved database (see next section).

4 EVER Diagrams for Specifying Schema Evolution

An *ER diagram* is a graphical representation of an ER database schema. In order to support the specification of changes to ER diagrams, we extend the basic graphical constructs of ER diagrams to present the relationships of schemas before and after a change (Figure 1). We call this diagram EVER diagram. In an EVER diagram, a designer can express the following associations:

- the derivation path of the new schema,
- the relationships of attributes between the new schema and the old schema,

- the participation of a new schema in relationship types (*i.e.*, edges in an ER diagram), and
- the conditions for maintaining invariant program views.

The derivation path indicates from where the new schema evolves. The attribute relationships specify the effect of changes to an attribute on other attributes, and can be expressed using functions. The change to an edge between an entity and a relationship type implies that the participation of the entity type in the relationship type needs to be established or dropped. Consequently, the relationship type needs to be evolved by adding to or deleting from the relationship type the key attribute of the affected entity type.

















G1		visible entity type	G8		defunct edge
G2		visible relationship type	G9		version derivation
G3		visible edge	G10		common, renamed resumed attribute
G4		key attribute	G11		domain changed attribute
G5		attribute	G12		derived attribute
G6		defunct entity type	G13		dependent attribute
G7		defunct relationship type	G14		resumed schema
G15		visible inheritance link	G16		defunct inheritance link

Fig. 1. The icons for EVER diagrams

Here we will use examples to illustrate the use of the extended graphical constructs (icons) G_1 to G_{14} (Figure 1). Icons G_{15} and G_{16} will be discussed in Section 6. Let us begin with the example shown in Figure 2(a). The new schema, V_2 , is derived from the old schema, V_1 . The derivation of the schema is represented using icon G_9 . Since the old schema cannot be seen by the new programs nor can be used for future evolution, we consider it as a defunct schema. A defunct entity type, relationship type and edge can be represented using icons G_6 , G_7 and G_8 , respectively. Therefore, V_1 is represented by a dotted rectangle. Similar to the defunct entity and relationship types, the resumed schema version which consists of the resumed attributes and all attributes of the old schema version can be represented using icons G_{14} .

The icons, from G_{10} to G_{13} , are used for representation of the attribute relationships. G_{10} indicates that the relationship of the two attributes at the two ends of the icon are common or one is renamed as the other. If the names of attributes at the two ends are the same, then they are common. Otherwise, the

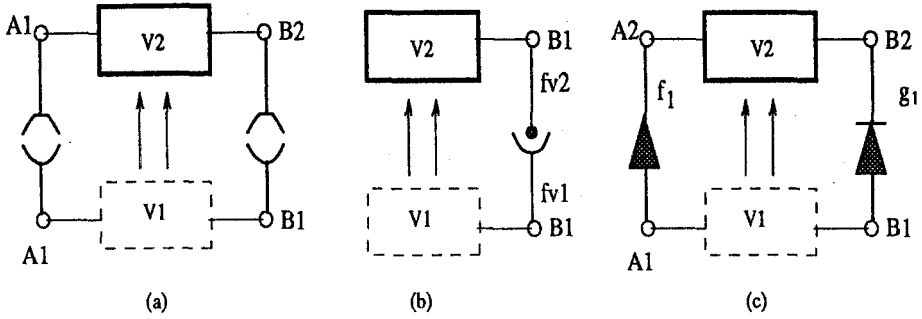


Fig. 2. The derivation of a schema the EVER diagram

attribute in the new version is renamed. For example, in Figure 2(a), attributes A_1 in the new and the old schema are common whereas attribute B_1 in the old schema is renamed as B_2 in the new schema. G_{11} is used for representation of a domain changed attribute. As shown in Figure 2(b), the domain of attribute B_1 in the new schema is different from that of attribute B_1 in the old schema, hence a forward function (f_{v2}) is associated with the end close to B_1 in the new schema. Similarly, a reverse function (f_{v1}) is associated with the end close to B_1 in the old schema. G_{12} and G_{13} are used for representation of a derived and dependent attribute, respectively. The attribute at the pointed end is derived from, or dependent on, the attributes in the other end. The derivation or prompt function for the attribute is associated with the attribute close to the pointed end. Finally, let us refer to Figure 2(c). Attribute A_2 in the new schema is derived from attribute A_1 in the old schema. Thus, a forward derivation function f_1 is associated with the pointed end of the icon close to attribute A_2 . On the other hand, attribute B_2 is dependent on B_1 and a prompt function (g_1) is associated with the pointed end of the icon close to B_2 .

As indicated in the previous section, in order to completely represent the attribute relationships among schema versions, a resumed schema version needs to be created to re-establish the relationships of eliminated attributes and therefore, a new schema version is always created based on the resumed schema version. Note that the resumed schema version is *virtual* in the sense that it does not physically exist. It serves as an aid to a designer for browsing the history of attribute relationships along the course of schema evolution. Through the EVER diagrams, a designer not only can visualize the relationships among schema versions but the designer can also browse through the history of changes.

At this point, let us revisit Table 3 and consider the specification of the change from V_2 to V_3 . From the table, one can see that attribute WarrantyExpires is newly added to V_3 , and the domain of attribute MPG is changed. Since there is an attribute relationship between attribute WarrantyBegins in V_1 and WarrantyExpires in V_3 , we must first create a resumed version, V'_2 in between V_2 and V_3 to accommodate the resume attribute WarrantyBegins and all the attributes of V_2 . Because of the resumed schema version, the relationship between attribute WarrantyExpires and WarrantyBegins can be re-established.

The attribute relationships between MPG in V_3 and V'_2 can be represented by derived functions. The reverse derivation function, f_1 , maps the domain of attribute MPG in the V_3 ($V_3(MPG)$) to that of the attribute in V'_2 ($V'_2(MPG)$), and the forward derivation function, f_2 , maps the domain of $V'_2(MPG)$ to that of $V_3(MPG)$. Attributes WarrantyBegins and WarrantyExpires can be derived from each other and can be specified together with functions f_1 and f_2 using LEVER (a Language for EVolutionary ER diagram [12]) as follows.

```

FUNCTIONS {
  ( $V_3(MPG) = f_2(V'_2(MPG))$ )
  WITH IMPLEMENTATION  $V_3(MPG) = itoa(V'_2(MPG))$ ;
  ( $V'_2(MPG) = f_1(V_3(MPG))$ )
  WITH IMPLEMENTATION  $V'_2(MPG) = itoa(V_3(MPG))$ ;
  ( $WarrantyExpires = f_3(WarrantyBegins)$ )
  WITH IMPLEMENTATION
     $WarrantyExpires = WarrantyBegins + 1$ ;
  ( $WarrantyBegins = f_4(WarrantyExpires)$ )
  WITH IMPLEMENTATION
    ( $WarrantyBegins = WarrantyExpires - 1$ ));

```

where $itoa()$ is a system function which converts an integer into a string.

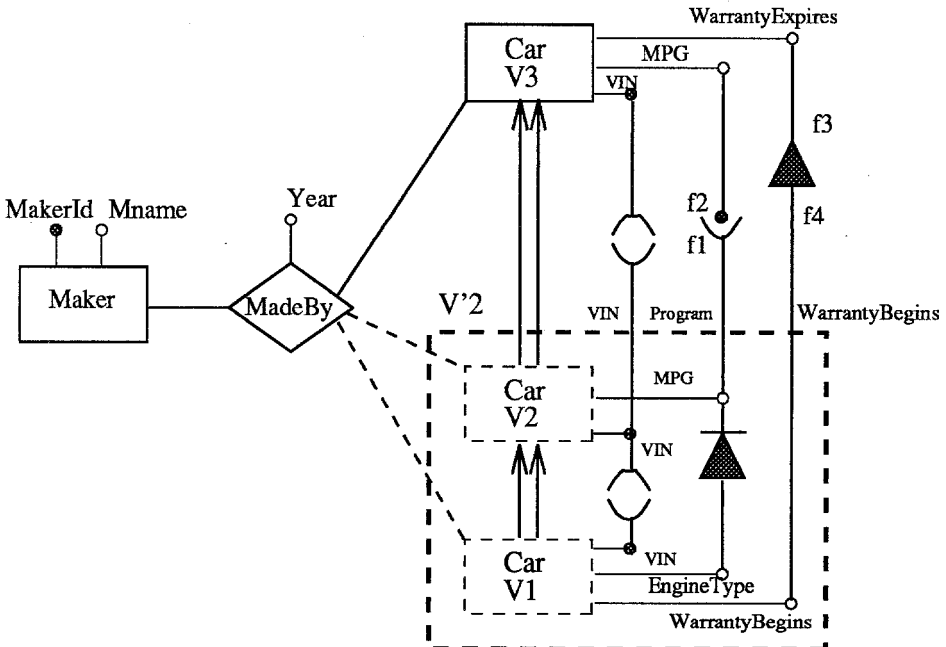


Fig. 3. An example of EVER diagrams for the specification of change

The EVER diagram representing the attribute relationships between schema version V_3 and V_2' is shown in Figure 3. The old schema version V_2 and the edge connecting to it are defunct, so they are represented by the dotted rectangle (G_6) and edge (G_8), respectively. Since V_3 evolves from V_2' which is virtual, it inherits all edges from V_2 . Schema version V_3 and edge that connects to it can be created and represented using a solid rectangle (G_1) and edge (G_3), respectively. The derivation of the new schema from the old one can be depicted by using a directed parallel line (G_9) which goes from V_2' to V_3 . More examples of the use of EVER diagrams to capture schema changes are shown in [11].

Thus far, we have discussed how EVER diagrams can capture all the aspects involved in the evolution of the schema of a database that does not support class hierarchies. Before considering schema evolution that involves class hierarchy and inheritance, in the next section we will show how EVER diagrams are translated into an underlying database model.

5 Transformation of EVER Diagrams into Databases

In order to support different implementation database models, instead of directly translating an EVER diagram into an underlying database schema, a VDG (*Version Derivation Graph*) representation, which is an internal representation of the EVER diagram, is created, and then map the VDG representation into a specific underlying data model. The VDG representation captures the semantics of the EVER diagram and provides the storage requirements of objects for mapping the EVER diagram into the specific underlying data model. In the VDG representation there is a set of VDGs. Each VDG represents the derivation of a particular schema in an EVER diagram. A VDG consists of a set of nodes and directed edges. A node captures the object structure (attributes) of a schema version and change relationships of the schema version to others. The objects created under a schema version are conceptually attached to the VDG node representing the schema version. A directed edge captures the object associations along the schema evolution course. There are two types of edges: derivation and inheritance edges. The inheritance edge will be discussed in Section 6. A derivation edge connects two VDG nodes corresponding to two schema versions (the old and the new) in an EVER diagram. That is, from the new schema version point of view, the edge indicates that the objects created under the old schema version must be converted to conform with the new schema version. Similarly, from the old schema version point of view, the objects created under the new schema version must be converted to conform with the old schema version. Thus, the objects associated with the VDG nodes connected by derivation edges (i.e., the *derivation lattice*) comprise the entire set of objects of the particular schema. The VDG representation is similar to the catalog and schemas with views to store the information about the mapping of EVER diagrams into logical databases. Since, a VDG is currently designed to support schema derivation, it is geared toward a single internal object representation. The schema of an object is conceptually represented as the union of attributes of all the schema versions in a derivation

lattice of the VDG (or the *complete schema*).

In considering the efficient maintenance of object consistency and use of storage among schema versions, when the underlying database is re-organized after a new schema version is created, objects are allocated additional storage for only those attributes (the *base attributes*) that cannot share the storage with attributes of the old schema. Let E_n be a schema version which is derived from schema versions E_1, E_2, \dots, E_m , where $n \notin \{1..m\}$. Attribute $a_i \in E_n$ is said to be a *base attribute* of E_n if and only if one of the following conditions are satisfied.

- $group(a_i) \in \{new, forward-dependent, forward-complementary-dependent\}$
- $\exists a_k \in E_j \wedge j \in \{1, \dots, m\}$, such that
 $a_i = domain_changed(a_k) \wedge (dom_size(a_k) \subset dom_size(a_i))$,

where $domain_changed(a)$ returns the attributes that are derived from attribute a and whose domain has been changed; $dom_size(a)$ computes the storage requirements for an attribute a . Let B_i be a set of base attributes of schema versions E_i , $i \in \{1..n\}$. The *complete schema* of schemas $\{E_1, E_2, \dots, E_n\}$ (S_c) can be expressed as: $S_c = \bigcup_{i=1}^n B_i$. Let us refer to the objects associated with a complete schema as *complete objects*. In order to indicate whether the objects created under a schema version need additional storage, we use two kinds of nodes in a VDG: *virtual* and *non-virtual* nodes.

A non-virtual node corresponds to a schema version which is either the initial one or is augmented with attributes that cannot be derived from the old schema. That is, a non-virtual node contains base attributes.

A virtual node corresponds to a schema which does not contain any base attribute.

Objects created under a new schema version that maps onto a non-virtual node cannot be stored in the underlying database described by the old schema versions. The underlying database needs to be re-organized in order to store objects created under the new schema version. On the other hand, the objects created from a schema version that maps onto virtual nodes can be completely stored in the underlying databases.

Let us demonstrate the transformation of VDGs into an implementation database schema which, we assume here, is relational (In the next section, we will show an example of a VDG transformation into an OODB schema). The relational database is "objectified" so that it can effectively support this mapping as well as the construction and use of database views representing the different schema versions. That is, we assume that each object, i.e. instance of entity or relationship type, is associated with a systemwide unique and immutable identifier (Oid) not visible to application programs.

Refer to the example shown in Figure 4 in which two schemas are merged together resulting in a new single schema. Suppose the schema Car initially consists of attributes {VIN, Color}, and class Bus consists of {VIN, Seats}. As

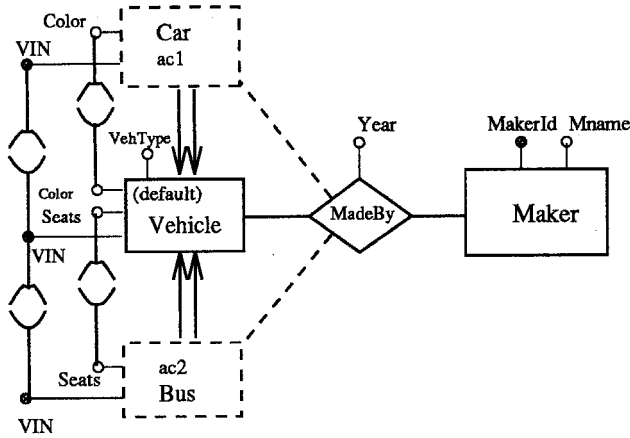


Fig. 4. An example of EVER diagrams for merging two schemas

indicated in the diagram, Vehicle is derived from schemas Car and Bus. Since Car and Bus are initial schemas, they are mapped into two VDGs, each consisting of a single non-virtual node, N_1 and N_2 , respectively. Being a non-virtual node, N_1 is mapped into a relation r_1 in the underlying database. The schema T_1 of r_1 contains all attributes of Car plus one extra attribute, the *object identifier* (Oid): $T_1(\text{VIN}, \text{Color}, \text{Oid})$. Similarly, the VDG node N_2 corresponding to Bus is mapped into a relation r_2 with schema $T_2(\text{VIN}, \text{Seats}, \text{Oid})$.

Schema Vehicle has three attributes: VehType (the type of a vehicle) which is a new attribute to Vehicle, VIN which shares with both Car and Bus, Color which shares with Car, and Seats with Bus. Since VehType as a new attribute, corresponds to a base attribute, Vehicle is mapped into a non-virtual VDG node, let us say N_3 . Thus, nodes N_1 , N_2 , and N_3 form a derivation lattice in the VDG. The complete schema (S_c) of the derivation lattice is the union of base attributes of Car, Bus and Vehicle: $S_c = \{\text{VIN}, \text{Seats}, \text{Color}, \text{VehType}\}$.

As in the case of VDG nodes N_1 and N_2 , being a non-virtual node, N_3 requires a new relation r_3 with schema T_3 ($T_3 = \{\text{VehType}, \text{Oid}\}$) to store the base attribute VehType. Since Vehicle shares attributes with Car and Bus schemas, Vehicle is represented as a relational view on r_1 , r_2 and r_3 .

In order to uniformly define a view for each schema version, we construct each view in terms of the complete schema. That is, objects associated with each schema version are expanded first to complete objects. In this example, the set of complete objects is the union of the set of the expanded objects associated with schemas Car, Bus and Vehicle. Each object schema, irrespective of whether it maps onto a virtual or non-virtual VDG node, is expressed as a view on the complete objects stored in the relations in the underlying database. Thus, the view of a schema version (S_i) is defined as a selection on the complete objects based on the access conditions associated with S_i , and then a projection on the attributes of S_i . Let *Expand()* be a procedure that converts an objects associated with a particular schema version to a complete object. The conversion of the

base attributes and the attributes viewed through the schema version make use of the functions specified in the EVER diagram. Let us illustrate step by step the construction of the views for schemas Car, Bus and Vehicle in Figure 4.

Step 1: Determine the complete schema of the derivation of the VDG. As indicated above, the complete schema (S_c) is $S_c = \{VIN, Seats, Color, VehType\}$.

Step 2: Determine the relations used to store the complete objects created by each schema version. In this example, schema Car is an initial schema and mapped into the schema of relation r_1 . Thus, the objects created under Car are stored into r_1 . Similarly, the objects created under Bus are stored into r_2 . However, the objects created under Vehicle must be stored into all three relations r_1 , r_2 and r_3 .

Step 3: Identify the complete objects created under a specific schema version. The objects created under a schema version may be stored in different relations. They can be identified by joining relations based on attribute Oid. For example, the objects created under schema version Vehicle (O_3^*) are selected by joining r_1 , r_2 and r_3 on Oid. Since the objects created under both Bus and Vehicle are stored in r_2 , we must separate them to apply the corresponding *Expand()* procedure. The objects created under version Bus (O_2^*) are selected by discarding the objects created under Vehicle from relation r_2 . Similarly, the objects created under Car are selected by removing the objects created under Vehicle from relation r_1 . The selection condition for identification of a set of the objects created under a schema version can be derived as the following table.

$schema(E_i)$	the created objects
Vehicle	$O_3^* = r_3 \bowtie_{Oid} r_2 \bowtie_{Oid} r_1$
Bus	$O_2^* = \sigma_{Oid \in (\Pi_{Oid}(r_2) - \Pi_{Oid}(O_3^*))}(r_2)$
Car	$O_1^* = \sigma_{Oid \in (\Pi_{Oid}(r_1) - \Pi_{Oid}(O_3^*))}(r_1)$

Step 4: Expand the complete objects created under a schema version to the objects viewed through the schema version, and then screen the objects that cannot satisfy the specified *conditions* out from the view of the programs. Suppose the programs that refer to entity types Car and Bus can be express as follows.

INVARIANT VIEWS {

(Car ACCESS WITH CONDITIONS ($ac_1 : VehType = car$));
 (Bus ACCESS WITH CONDITIONS ($ac_2 : VehType = bus$)));

Let $View_i$ represent the view for schema E_i . If there are n schema versions, then, the view of a schema version can be defined uniformly as belows:

$$View_i = \Pi_{E_i}(\sigma_{Conditions_{E_i}}(\bigcup_{i=1}^{i=n} Expand(O_i^*))),$$

where Π stands for projection, σ for selection and $Conditions_{E_i}$ for the conditions specified against E_i . Therefore, in the example, the view of each schema version can be expressed as:

$$\begin{aligned} View_{Car} &= \Pi_{\{VIN, Color\}}(\sigma_{\{VehType=car\}}(\bigcup_{i=1}^{i=3} Expand(O_i^*))) \\ View_{Bus} &= \Pi_{\{VIN, Seats\}}(\sigma_{\{VehType=bus\}}(\bigcup_{i=1}^{i=3} Expand(O_i^*))) \\ View_{Vehicle} &= \Pi_{\{VIN, Color, Seats, VehType\}}(\bigcup_{i=1}^{i=3} Expand(O_i^*)) \end{aligned}$$

Each view is stored in the corresponding VDG node and it may need to be reconstructed after each database re-organization.

In our approach, we can guarantee that the update against a view can be correctly translated into the sequence of updates on the complete objects in the underlying database because of the following two reasons: (1) the key attributes of different schema versions must be same or the mapping among them must be one-to-one. Therefore, the complete objects stored in the underlying database can be uniquely identified by using different key attributes. (2) The objects viewed from a schema version are always a subset of the complete objects, and can be mapped into the unique complete objects in the underlying database.

6 Evolution of Extended ER Diagrams

The presentation of the EVER diagrams thus far has been restricted to the evolution of database schemas for traditional database applications which can be modeled using classification and aggregation mechanisms. However, in some systems, such as object-oriented databases, there may exist *inheritance relationships* among entity types that capture the generalization/specialization abstraction between *superclasses* and *subclasses* of entity types. In the ER model, class hierarchy and inheritance can be expressed using *extended ER* (EER) diagrams [9, 21]. In this section, we discuss how EVER diagrams can facilitate the evolution of EER diagrams and can be mapped into an OODB schema.

A. The Evolution of Inheritance Lattices

When a class in an *inheritance lattice* evolves, we require that the entire inheritance lattice is evolved. Let us refer to the versions of an inheritance lattice before and after evolution as the old and new lattices, respectively. Similar to the evolution of an entity type (see Section 4), the resumed schema of an inheritance lattice can be considered as the union of all the attributes of the classes in the old lattice plus all the eliminated attributes of the previous versions of the inheritance lattice. Once the resumed schema of the lattice is computed, the attribute relationships between the new lattice and the resumed lattice, and the conditions for maintenance of a consistent database can be specified using EVER diagrams.

In addition to the ER/EER constructs, the EVER diagram provides a construct, namely, the defunct inheritance links (Icon G_{16} as shown in Figure 1), that allows a designer to specify changes to inheritance relationships between subclasses and superclasses in an EER diagram:

1. Add a new class and its associated inheritance links to an inheritance lattice.
2. Drop a class (with no objects) and its associated inheritance links from an inheritance lattice, possibly in conjunction with a reduction of the domain of the defining attribute which is used to define an entity type as a subclass in an inheritance lattice.
3. Add to or delete inheritance links from an inheritance lattice.

4. Change the inheritance ordering (i.e., reverse the position of two classes in an inheritance lattice).

In order to maintain a consistent database for application programs after a change, our approach requires that changes must satisfy the criteria of *object consistency*, *key consistency*, and *invariant program views* (see Section 3).

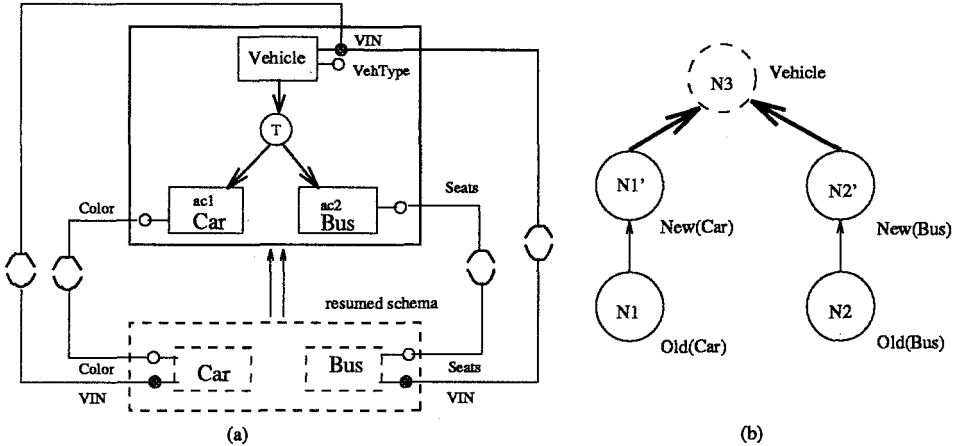


Fig. 5. (a) An evolved inheritance lattice in an EVER diagram, and (b) the corresponding VDG

Let us use the example shown in Figure 5(a) to demonstrate how the EVER is used for specifying the evolution of an inheritance lattice. Assume that the class *Vehicle* is generalized from the two classes *Car* and *Bus* (Figure 4). The class *Vehicle* gains a new attribute *VehType* indicating the type of Vehicles. Thus, the resumed schema (R_s) of the lattice is the union of the attribute sets of classes *Old(Car)* (the old version of class *Car*) and *Old(Bus)* (the old version of class *Bus*). That is, $R_s = \{VIN, Color, Seats\}$. Out of these attributes, *VIN* is common to all three classes *Vehicle*, *Car* and *Bus*. Attribute *VehType* is new to the class *Vehicle*. Assume that the default value of attribute *VehType* is *car* if objects are associated with class *Car*, and it is *bus* if objects are associated with class *Bus*. Thus, the specification of *default* functions can be expressed as follows.

FUNCTION { ((*VehType*(x) = *default*)
WITH IMPLEMENTATION
 (if *Schematype*(x) = *Car* then *VehType*(x) = *car*
 else *VehType*(x) = *bus*)) };

B. The Mapping of an EVER Diagram onto VDGs

In the case of an inheritance lattice, a single VDG is used to represent all the entity types (classes) which are part of the inheritance lattice. Such a VDG

can be constructed as follows. First, we compute the attribute set of each class in the new lattice, and then, compare it with that of its corresponding old class if there exists such one. If the new and old attribute sets are the same, both the old and new classes share the same VDG node. On the other hand, if the new class owns base attributes, then a non-virtual node is created for it; otherwise, a virtual node is created for that new class. Last, for each new node representing a class in the new lattice, establish an derivation path from a node corresponding to a class from which the new class has evolved.

Consider again the above example shown in Figure 5(a). The EVER diagram can be mapped onto a VDG (Figure 5(b)) in the following way. Since class Vehicle has no corresponding class in the old inheritance lattice, and its participation constraint is *total*, it is mapped to a virtual node N_3 (dotted). Classes Car and Bus gain an additional attribute by inheriting VehType from Vehicle. Hence, both Car and Bus are mapped onto two new non-virtual nodes N'_1 (New(Car)) and N'_2 (New(Bus)), respectively. The edges from N_1 to N'_1 and from N_2 to N'_2 are derivation edges (thin lines) because New(Car) evolves from Old(Car) and New(Bus) from Old(Bus). However, the edges from N'_1 to N_3 and from N'_2 to N_3 are inheritance edges (thick lines) because they are established based on the inheritance links between Vehicle and New(Car) and between Vehicle and New(Bus), respectively.

C. The Mapping of a VDG into an OODB

The mapping of a VDG into a relational database and the construction of views for each version of a class in an inheritance lattice can be handled by using a similar approach presented in Section 4. Below, we will demonstrate the potential that a VDG can be mapped into an OODB.

In order to implement the EER schema evolution in the context of an OODB, the primitive *Augment()* is introduced to augment an object structure in the underlying OODB database originally defined by an old class. For example, as shown in Figure 5(b), since the VDG nodes N'_1 (New(Car)) and N'_2 (New(Bus)) are non-virtual VDG nodes evolved from nodes N_1 (Old(Car)) and N_2 (Old(Bus)), respectively, the underlying object structures for Old(Car) and Old(Bus) need to be augmented: *Augment*(Old(Car)) with {VehType}, and *Augment*(Old(Bus)) with {VehType}. Since the node N_3 representing class Vehicle is virtual, and has inheritance edges connecting to subclasses N'_1 (New(Car)) and N'_2 (New(Bus)), we do not create an object structure for the class Vehicle. The objects created under the class Vehicle are either stored into the augmented class Car or Bus. In other words, the objects associated with class Vehicle is the union of the objects associated with augmented classes Car and Bus.

In this example, the complete schema (S_{Car}) of derivation lattice Car consists of {VIN, VehType, Color}, and the corresponding complete objects (O_{Car}) are stored in the augmented class Car. Similarly, the complete schema (S_{Bus}) of derivation lattice Bus consists of {VIN, VehType, Seats}, and the corresponding complete objects (O_{Bus}) are stored in the augmented class Bus. The objects of the class Vehicle are the complete objects of classes Car and Bus ($O_{Car} \cup O_{Bus}$). Thus, the view for each version of the VDG can be constructed as follows.

$$\begin{aligned}
View_{Vehicle} &::= \Pi_{\{VIN, VehType\}}(\{x \cup y \mid x \in O_{Car}; y \in O_{Bus}\}) \\
View_{Old(Car)} &::= \Pi_{\{VIN, Color\}}(\{x \mid x \in O_{Car}\}) \\
View_{New(Car)} &::= \Pi_{\{VIN, VehType, Color\}}(\{x \mid x \in O_{Car}\}) \\
View_{Old(Bus)} &::= \Pi_{\{VIN, Seats\}}(\{y \mid y \in O_{Bus}\}) \\
View_{New(Bus)} &::= \Pi_{\{VIN, VehType, Seats\}}(\{y \mid y \in O_{Bus}\})
\end{aligned}$$

These views reflect the schema of the new inheritance lattice which is mapped into the following object-oriented database schema (using C++ syntax):

```

class Vehicle { char * VIN; char * VehType; }
class Car : Vehicle { integer Color; }
class Bus : Vehicle { integer Seats; }

```

7 Conclusion

In this paper, we presented a schema derivation approach to schema evolution through changes to the ER schema of a database. The approach is supported by EVER, an EVolutionary ER diagram for specifying the derivation relationships between schema versions in ER/EER diagrams, relationships among attributes, and the conditions for maintaining consistent views of programs. A methodology was presented for mapping the EVER diagram into the underlying database and constructing database views for schema versions. Through the reconstruction of views after database reorganization, changes to an ER/EER diagram can be made transparent to the application programs while all objects in the database remain accessible to the application programs. In order to demonstrate its potentials, we used our methodology to evolve two database schemas, one that assumed that the underlying database is relational, and the second that assumed that the underlying database is object-oriented.

Since an EVER diagram may become quite complex in the case of large applications, we are investigating an icon-based EVER diagram, in which a sub-diagram can be encapsulated and replaced by an icon (called a complex icon). This will lead to a cleaner and less clogged visual specification. In order to evaluate how well the proposed approach scales up, we plan to use it on real world applications and study empirically its effects, in a manner similar to Sjöberg's work [18] that quantifies schema evolution.

Finally, we intend to build a prototype for exploration of schema evolution in different database models, and also to experiment with schema integration in an attempt to facilitate interoperability in a multidatabase system.

References

1. M. Ahlsen et al. Making Type Changes Transparent. In *Proc. of IEEE Workshop on Language for Automation*, 1983.
2. J. Banerjee et al. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. In *Proc. of ACM SIGMOD*, 1987.

3. E. Bertino. A View Mechanism for Object-Oriented Databases. In *Proc. of 3rd Intl. Conference on Extending Database Technology*, 1992.
4. S. E. Bratsberg. Unified Class Evolution by Object-Oriented Views. In *Proc. of the 11th Intl. Conference on Entity-Relationship Approach*, 1992.
5. R. Bretl et al. The GemStone Data Management Systems. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley Publishing Co., 1989.
6. P. Chen. The Entity Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), 1976.
7. P. Chen. ER vs. OO. In *Proceedings of the 11th International Conference on Entity-Relationship Approach*, 1992.
8. S. M. Clamen. Schema Evolution and Integration. *Journal of Distributed and Parallel Databases*, 1994.
9. R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*, 2nd edition. The Benjamin/Cummings Publishing Company, Inc., 1992.
10. L. Mark, Litwin, W. and N. Roussopoulos. Interoperability of Multiple Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.
11. C. T. Liu, S. K. Chang, and P. K. Chrysanthis. Database Schema Evolution using EVER Diagrams. In *Proc. of Intl. Workshop on Advanced Visual Interfaces*, 1994.
12. C.T. Liu, P.K. Chrysanthis, and S.K. Chang. Schema Evolution through Changes to ER Diagrams. *Journal of Computer and Information Sciences*, 9(4), 1994.
13. S. B. Navathe and M. K. Pillalamarri. OOER: Toward Making the E-R Approach Object-Oriented. In *Proc. of the 8th Intl. Conference on Entity-Relationship Approach*, 1989.
14. X. Qian and G. Wiederhold. Knowledge-based Integrity Constraint Validation. In *Proc. of the Intl Conference on Very Large Data Bases*, 1986.
15. M. E. Segal and O. Frieder. On-the-Fly Program Modification: Systems for Dynamic Updating. *IEEE Software*, 1993.
16. A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3), 1990.
17. E. Simon and P. Valduriez. Design and Implementation of an Extendible Integrity Subsystem. In *Proc. of ACM SIGMOD*, 1984.
18. Dag Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1), 1993.
19. H. A. Skarra and S. B. Zdonik. Type Evolution in an Object-Oriented Database. *Research in Object-Oriented Databases*. Addison-Wesley, 1987.
20. M. Stonebraker and L. A. Rowe. The Design of POSTGRES. In *Proc. of ACM SIGMOD*, 1986.
21. T.J. Teorey, D. Yang, and J.P. Fry. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model. *ACM Computing Survey*, 18(2), 1986.
22. Thomas et al. Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys*, 22(3), 1990.
23. S. B. Zdonik. Object-Oriented Type Evolution. *Advances in Database Programming Languages*. Addison-Wesley, 1990.
24. R. Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *Proc. of Conference on Data Engineering*, 1991.