Database Schema Evolution using EVER Diagrams

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

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

Abstract

We present an approach to schema evolution through changes to the ER diagram representing the schema of a database. In order to facilitate changes to the ER schema 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. In this paper, we demonstrate the mapping of the EVER diagram into an underlying database and the construction of 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. The EVER system can serve as a front-end for object-oriented databases.

1 Introduction

In this paper we present an approach to schema evolution through changes to the ER diagram representing the schema of a database [5]. In order to facilitate changes to the ER schema 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. We also describe the mapping of the EVER diagram into an underlying database and the construction of 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.

Our approach is illustrated in Figure 1. The user interface of the EVER system allows the application programmer to create and modify the EVER diagram. The EVER diagram serves as the visualization aid that graphically conveys changes to a database schema. The diagram is then transformed to an intermediate representation called *version derivation graphs* (VDGs) which are subsequently mapped into the structures of an underlying database. A powerful visual interface is thus provided for database schema evolution.

Various approaches to database schema evolution have been proposed, particularly in the context of object-oriented databases [3, 4, 7, 2, 13, 15, 16]. Theses approaches can be divided into three categories based on the external representation of the structure of the objects in the database (object schema) to application programs and interactive users, and the internal representation of the objects in the underlying database: (1) *Schema modification* approaches [2, 16] *always support one single schema and a single internal representation for each object.* Hence, all objects must be con-

*This material is partially supported by the National Science Foundation under grant IRI-9210588.

verted 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. (2) *Schema versioning* approaches [1, 13] support multiple schemas and multiple internal object representations for an object. 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 must always stay in that version. Therefore, 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.* (3) *Schema derivation* approaches [3, 4, 7, 15] 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 the common representation. The instantiation of objects to a schema version is performed at run-time. 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 and, as such, supports multiple schema versions and a common internal object representation, referred to as the *complete object.* However, in our approach a designer can only make changes to the up-to-date schema. Therefore, conflicts with previous schema versions are avoided. Although EVER is designed mainly to support an approach for schema derivation, it can also be extended to support other two schema evolution approaches. It should be pointed out that the ER diagrams have been extended in the past as we do here with EVER diagrams in order to incorporated new concepts. Examples include the *Extended ER* (EER) model that incorporates the class hierarchy [9, 14] and the *Concept D*, a graphical language that supports multilevel concept structures [10].

The rest of the paper is organized as follows. In Section 2, we will analyze the attribute relationships among schema versions and discuss the issues in maintaining a consistent database which motivate the need for EVER

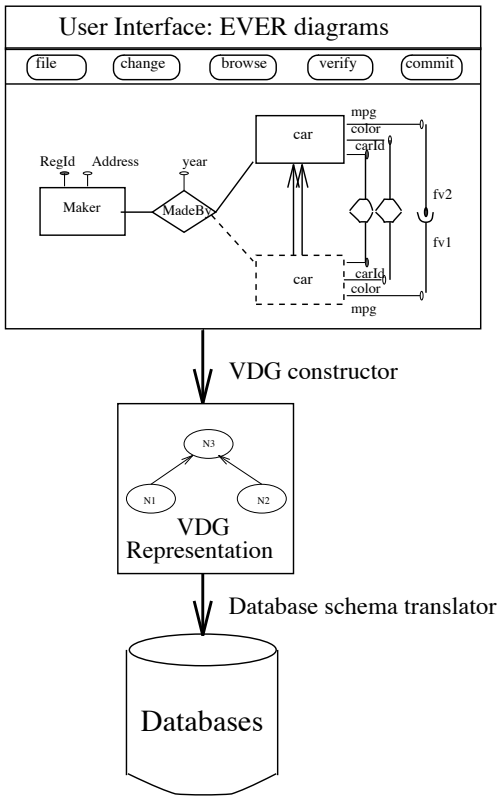

Figure 1: The overview of the EVER System for schema evolution

diagrams. Section 3 introduces the extended graphical constructs for expressing changes to ER diagrams, and then present several examples of EVER diagrams. In Section 4 we describe a methodology for the transformation of EVER diagrams into the underlying databases which are assumed to be relational [8]. In the concluding section, we argue that the EVER system can be used as a front-end for object-oriented databases, and outline the future work.

2 Schema Evolution through Changes to ER Diagrams

The approach proposed in this paper supports the transparency of changes to a schema for the existing application programs while facilitates the requirements of new applications. When an entity or relationship type (or schema for short) is changed, a new version of the schema is then created. Each schema version is the interface for programs to access the database.

A. Analysis of Attributes in Different Schema-Versions

When a schema evolves, the most important relationship between the old and the new schemas is the relationships of their attributes. These relationships provide the crucial information for reorganization of the objects in the underlying database and maintaining object consistency. We classify the attributes between two schemas based on the relationships of their values, their domains and their names along similar lines as in [7].

- *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 the name of the attribute in the two schemas is exactly 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 an early schema version but it is added back to a latter schema version. A resumed attribute can be handled in the same way as a common attribute.
- *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 say B, is said to be *dependent* if the value of the attribute is affected by changes to the values of other attributes, let say $\{A_1, A_2, \cdots, A_k\}$, but the value of the dependent attribute cannot be *derived* from the values of the same attributes $\{A_1, A_2, \cdots, A_k\}$.
- *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 $\{A_1, A_2, \cdots, A_k\}$ are attributes of the old schema, and B is the attribute of the new schema, then attribute B is classified into the **forward** group. If $\{A_1, A_2, \cdots, A_k\}$ are attributes of the new schema, and B is the attribute of the old schema, then attribute B is in the **reverse** group. If $\{A_1, A_2, \cdots, A_k\}$ can be attributes in the new schema or old schemas, and B is an attribute of the new schema, then B is classified into **forward complementary** group. However, if B is an attribute of the old schema, then B is in **reverse complementary** group.

B. Explicit Specification of Attribute Relationships

The attribute relationships can be expressed by using the following four general functions.

Identity function. If attributes a and b are common, their relationship can be represented by using an *identity function* (I) such that $a = I(b)$, or $a \equiv b$.

Derivation function. If attribute a can be derived from only attributes $b_1, b_2, \cdots b_k$, the relationship of a to attributes $b_1, b_2, \cdots b_k$ can be represented using a *derivation function* (F) such that $a = F(b_1, b_2, \cdots b_k)$.

Prompt function. If attribute a depends on attributes $b_1, b_2, \cdots b_k$ but it cannot be derived solely from $b_1, b_2, \cdots b_k$ (*e.g.*, it may need additional information), the relationship of a to attributes $b_1, b_2, \cdots b_k$ can be represented by using a *prompt function* (Ψ) such that $a = \Psi(b_1, b_2, \cdots b_k, \Phi)$, where Φ represents the additional information. Φ is possibly an interactive query against the rest of the database that is not involved in the particular schema changes.

Default function. If the attribute value of a of an object is unspecified but the value is required by a program, then the *default* value can be acquired by

using a *default function* (*default*). By assigning a default value to a unspecified attribute value, the need of the application programs associated with different schema versions can be resolved.

In order to indicate the mapping direction of a function, we can prefix the *forward* or *reverse* to the function. *Forward* indicates the mapping is from the old to new schema version, and *reverse* indicates the mapping is from the new to old schema version. Except for resumed attributes, attribute relationships can only explicitly exist in two consecutive schema versions. An attribute of a schema version V_i can only be resumed in another V_l, $i < l$, if there is no such attribute in any schema version V_j in between V_i and V_l. Thus, attribute relationships may exist between V_i and V_l which are not necessarily consecutive. The resumed attribute allows for capturing these types of relationships.

C. The Maintenance of Database Consistency Across Schema-Versions

A database is said to be consistent if and only if for each state of an object in the database, two observers view the same state through different schema versions at any time, the result must agree on 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. The maintenance of object consistency can be accomplished through the functions discussed in the previous section. Whenever the value of an attribute of an object is updated, those attributes depending on the updated attribute are also updated based on the specified functions. An update of an attribute and the propagation of the update to the affected attributes are executed as a transaction.

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 schema version.* However, the evolved database may not preserve the interpretations made by the programs associated with the previous schema versions. In our framework of schema evolution, we provide facilities to allow the designer to specify the conditions under which the programs can maintain their consistent views to the evolved database.

3 EVER Diagrams for Specifying Schema Evolution

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. We call this diagram EVER diagram. In an EVER diagram, a designer can express the following relationships:

- the evolution relationship of the new schema,
- the relationships of attributes between the new schema and the old schema,
- the relationship of a new schema (*i.e.*, edges) to the other schemas, and
- the invariant views of programs to the database.

The evolution relationship indicates from where the new schema evolves. The attribute relationships specify the effect of changes to an attribute on the others, and can be represented by 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. And consequently, the relationship type needs to be evolved by

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

Figure 2: The icons for EVER diagrams

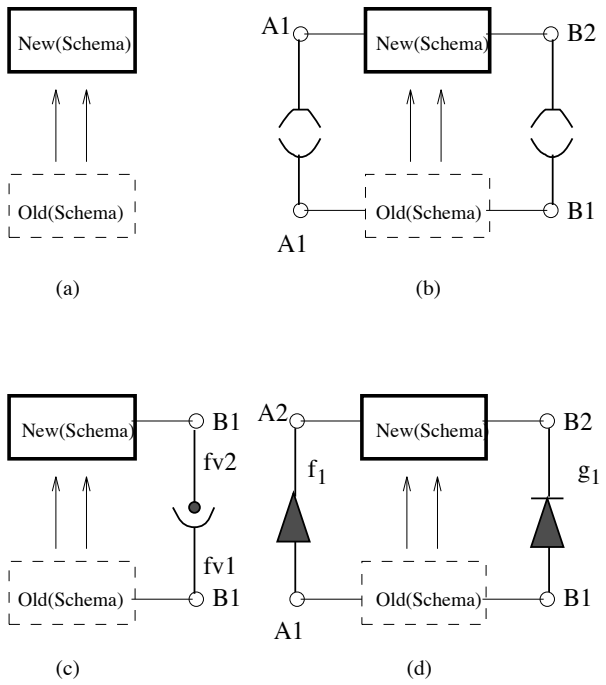

Figure 3: The derivation of a schema the EVER diagram

adding to or deleting from the relationship type the key attribute of the affected entity type. The conditions for maintenance of invariant program views ensure that the programs can access the evolved database consistently. The conditions for maintenance of invariant program views ensure that the programs can access the evolved database consistently.

The extended graphical constructs (icons) are shown in Figure 2. We will use examples to illustrate the uses of the icons. Let us begin with the one shown in Figure 3(a). The new schema, New(Schema), is derived from the old schema, Old(Schema). The derivation of the new schema is represented using icon G_9 (a parallel directed line). Since the old schema cannot be seen by the new programs, we consider it as a defunct schema. Thus, Old(Schema) is represented by a dotted rectangle. Similar to the defunct schema, the resumed schema version which consists of the resumed attributes and all attributes of the old schema version can be represented using icon 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. For example, in Figure 3(b), attribute A_1 in the new schema and attribute A_1 in the old schema are common. However, 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. The forward function is associated with the end close to the attribute in the new schema version, and the reverse function is associated with the end close to the attribute in the old schema version. As shown in Figure 3(c), the domain of attribute B_1 in the new schema is different from that of attribute B_1 in the old schema. Therefore, the forward function (f_{v2}) is associated with the end close to attribute B_1 in the new schema. Similarly, the reverse function (f_{v1}) is associated with the end close to the attribute 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 at-

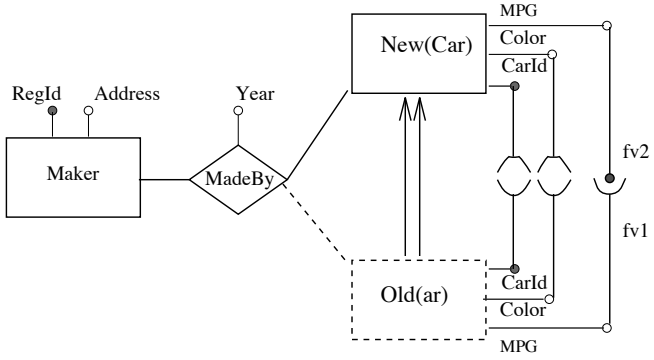

Figure 4: An example of EVER diagrams for the specification of domain change

tribute close to the pointed end. Let us refer to Figure 3(d). Attribute A_2 in the new schema is derived from attribute A_1 in the old schema. The 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. Thus, the prompt function (g_1) is associated with the pointed end of the icon close to the attribute B_2.

Thus far, we have discussed how the icons used in EVER diagrams can capture all the aspects involved in the evolution of a database schema. In the following two examples, we will illustrate the diagrammatical representation of a specification of changes to an ER diagram. We assume that all the changes must satisfy the constraints in maintaining the structurally consistent ER diagrams and the consistent database.

The first example, as shown in Figure 4, illustrates a change to the domain of attribute, MPG, mileage per gallon, of schema Car. The domain of attribute MPG is changed from MPG: integer[0 .. 9999] to MPG: string[10]. All other attributes in the new and old schema versions remain unchanged. The mapping between the new and old domains can be supported by functions, *itoa*() and *atoi*(), which are supported by the system. Functions atoi() and itoa() are used to convert a string into an integer and an integer to a string, respectively. Therefore, the attribute relationships can be represented by derived functions. The reverse derivation function (f_{v1}) maps the domain of attribute MPG in the new schema (New(MPG)) to that of the attribute in the old schema (Old(MPG)), and the forward derivation function, (f_{v2}), maps the domain of Old(MPG) to that of New(MPG).

In this EVER diagram, attributes RegId, Color are common to both schema versions. Their relationships are represented by using icon G_{10}. Attribute MPG are domain changed attribute. Thus, the relationship between New(MPG) and Old(MPG) is represented using icon G_{11}. The forward and reverse derivation functions (f_{v2} and f_{v1}, respectively) are associated with the ends close to attributes New(MPG) and Old(MPG) , respectively. Functions f_{v2} and f_{v1} can be specified as follows.

FUNCTIONS {
$(New(MPG) = f_{v2}(Old(MPG);$
WITH IMPLEMENTATION
$New(MPG) = itoa(Old(MPG)));$
$(Old(MPG) = f_{v1}(New(MPG))$
WITH IMPLEMENTATION
$Old(MPG) = itoa(New(MPG)))\};$

The new schema, New(Car), 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 the old schema to the new one. The old schema, Old(Car), and the edge connecting to it are defunct, and can be represented by the dotted rectangle (G_6) and edge (G_8), respectively, and they are not visible to the programs associated with the new schema any more.

In the second example, let us demonstrate an EVER diagram (as shown in Figure 5) in which two schemas are merged together resulting in a new single schema. As indicated in the diagram, Company is derived from schemas Maker and Dealer. Since Maker and Dealer have the common key attribute RegId, the new entity type, Company, inherits the key attribute, and gains an additional attribute CompanyType to distinguish the type of a company. Attribute CompanyType is new because it is independent with respect to the schemas Maker and Dealer. The default value of the attribute can be defined as:

CompanyType(x)

$$= \begin{cases} dealer & \text{if } Schema_{type}(\text{x}) = \text{Dealer} \\ maker & \text{if } Schema_{type}(\text{x}) = \text{Maker} \end{cases}$$

Function $schema_{type}$() takes an object as its input, and returns the name of the schema to which the ob-

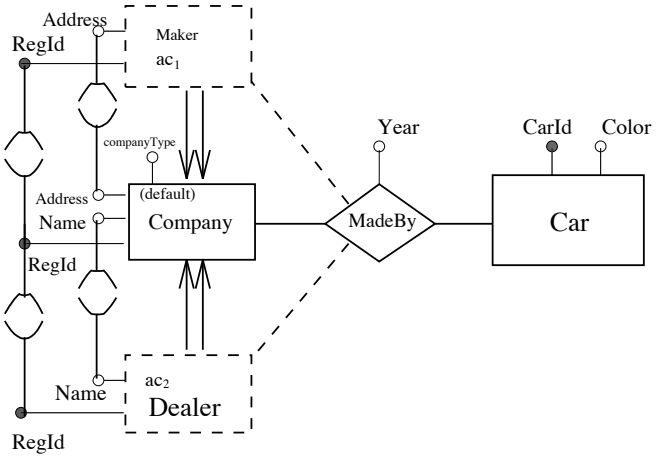

Figure 5: An example of EVER diagrams for merging two schemas

ject belongs. The programs that use the old schema may need to access a part of the evolved database. For example, the programs that refer to entity type Maker may just need to access the objects whose value of CompanyType is equal to *maker*. The default values and the conditions used for maintaining the invariant views of the programs associated with a schema version are specified as follows.

FUNCTIONS {
  $((companyType(x) = default)$
  WITH IMPLEMENTATION
    $(if\ Schema_{type}(x) =$ Dealer
    $then\ companyType = dealer$
    $else\ companyType = maker))$ };

INVARIANT VIEWS {
  (Maker **ACCESS WITH CONDITIONS**
    $(ac_1 : CompanyType = maker)$);
  Dealer **ACCESS WITH CONDITIONS**
    $(ac_2 : CompanyType = dealer))$};

In the EVER diagram, the default functions for an attribute are associated with the attribute, and the view conditions for a schema version are associated with that schema version. The resultant EVER diagram shown in Figure 5.

4 Transformation of EVER Diagrams into Databases

In order to support different implementation database models, instead of directly translating an EVER diagram into the underlying database model, our approach transforms the EVER diagram into a conceptual representation called the *version derivation graphs* (VDGs), and then maps the VDGs into the underlying database model.

A VDG captures the evolution of a particular schema. It consists of a set of nodes and directed edges. Each node corresponds to a schema version recording the attribute relationships to the previous and the following schema versions and the conditions for maintaining object consistency. When a new schema version is specified in an EVER diagram, a new node representing the new schema version in added in the corresponding VDG. A directed edge represents the derivation relationship among schema versions. 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 in the VDG as the union of attributes of all the versions of the schema (or the *complete schema*).

In considering the efficient maintenance of object consistency and use of storage among schema versions, when the underlying database is reorganized 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 version. Let E_n be a schema version which is derived from schema versions $E_1, E_2, \cdots, E_m$, 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* $\}$
- $\exists a_k \in E_j \wedge j \in \{1, \cdots, m\}$, such that $a_i = domain\text{-}changed(a_k) \bigwedge (dom\_size(a_k) \subset dom\_size(a_i))$.

where $dom\_size(a)$ is a function used to compute the storage for an attribute a; domain-changed(a) returns the attributes that is derived from attribute a, but whose domain has been changed. Let B_i be a set of base attributes of schema versions E_i, $i \in \{1..n\}$. The *complete schema* of schemas $\{E_1, E_2, \cdots, E_n\}$ (S_c) can be expressed as: $S_c = \bigcup_{i=1}^{n} B_i$. Let us refer to the objects correspond to the complete schema as the *complete objects*.

In order to indicate whether the objects created un-

der a schema version need additional storage, we define two kinds of nodes: *virtual* and *non-virtual* nodes.

A non-virtual node corresponds to an schema version which is either the initial one or is augmented with the 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 schema that maps onto a non-virtual node cannot be stored in the databases described by the old schema versions. Thus, the underlying database need to be re-organized. On the other hand, the objects created from a schema that map onto virtual nodes can be completely stored in the underlying databases.

The representation of changes to an ER diagram using VDGs provides the independence from the underlying database model. We will demonstrate the transformation of VDGs to an implementation database schema which, we assume to be relational. 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.

To illustrate the mapping from an EVER diagram into the relational database, let us use the example shown in Figure 5. As indicated in the diagram, Company is derived from schemas Maker and Dealer. Since Maker and Dealer are initial schemas. They contain base attributes, and thus are mapped into two VDGs. Each consists 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 whose schema T_1 contains all attributes of Maker plus one extra attribute, the object identifier Oid: T_1(RegId, Address, Oid). Similarly, Dealer maps to VDG node N_2, and then maps to a relation r_2 with schema T_2(RegId, Name, Oid).

Schema Company has three attributes: CompanyType which is a new attribute to Company, RegId which shares with both Maker and Dealer, Address which shares with Maker, and Name with Dealer. Since CompanyType is a base attribute, schema Company is mapped into a non-virtual node, N_3. The complete schema (S_c) of the VDG with nodes N_1, N_2 and N_3 is the union of base attributes of Maker, Dealer and Company: S_c = {RegId, Name, Address, CompanyType}. 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(CompanyType, Oid) to store the base attribute Companytype. Thus, Company is represented as a 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 Maker, Dealer and Company. 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. 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 object 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 Maker, Dealer and Company in Figure 5.

Step 1: Determine the complete schema of the VDG. As indicated above, the complete schema (S_c) of schema Car is {RegId, Name, Address, CompanyType}.

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

Step 3: Identify the objects created under a specific schema version, and expand them into the complete objects. The objects created under a schema version

may be stored in different relations. They can be identified by joining relations based on Oid. For example, as shown in the following table, the objects created under schema version Company (O_3) are selected by joining r_1, r_2 and r_3 on Oid. Since the objects created under both Dealer and Company are stored in r_2, we must separate them to apply the corresponding *Expand*() procedure. The objects created under version Dealer (O_2) are selected by discarding the objects created under Company from relation r_2. Similarly, the objects created under Maker are selected by removing the objects created under Company from relation r_1.

schema	the created objects
Company	$O_3 = r_3 \bowtie_{Oid} r_2 \bowtie_{Oid} r_1$
Dealer	$O_2 = \sigma_{\mathrm{Oid} \in (\Pi_{\mathrm{Oid}}(\mathrm{r_2}) - \Pi_{\mathrm{Oid}}(\mathrm{O_3}))}(\mathrm{r_2})$
Maker	$O_1 = \sigma_{\mathrm{Oid} \in (\Pi_{\mathrm{Oid}}(\mathrm{r_1}) - \Pi_{\mathrm{Oid}}(\mathrm{O_3}))}(\mathrm{r_1})$

Step 4: Construct a view for a schema version. Convert 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. Let $View_i$ represent the view for schema S_i. If there are n schema versions, then, the view of a schema version can be defined uniformly as belows:

$$View_i = \Pi_{S_i}(\sigma_{\mathrm{Conditions_{S_i}}}(\bigcup_{i=1}^{i=n} Expand(O_i)))$$

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

$$View_{Maker} = \Pi_{\langle RegId, Address \rangle}(\sigma_{\langle CompanyType = maker \rangle}(\bigcup_{i=1}^{i=3} Expand(O_i)))$$

$$View_{Dealer} = \Pi_{\langle RegId, Name \rangle}(\sigma_{\langle CompanyType = dealer \rangle}(\bigcup_{i=1}^{i=3} Expand(O_i)))$$

$$View_{Company} = \Pi_{\langle RegId, Address, Name \rangle}(\bigcup_{i=1}^{i=3} Expand(O_i))$$

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 based on the following reasons.

- The key attributes of different schema versions must be same or the mapping among them must be one-to-one. Therefore, the objects viewed from a schema version (view objects) can always be mapped into the unique complete objects in the underlying database.
- The objects viewed from a schema version (view objects) are always a subset of the complete objects, and can be mapped into the unique complete objects in the underlying database.
- The functions used for representation of attribute relationships indicate a unique way to translate the view update into the updates against the underlying database.

5 Conclusion

This paper presented a graphic specification language to support schema evolution based on the Entity-Relationship (ER) approach for data modeling. We chose to examine the semantics of changes in the context of the ER model for the following reasons. Firstly, this approach has the advantages of being graphic oriented and of being 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, the ER model supports many types of relationships whereas Object-Oriented models primarily support one type of relationship, which is similar to the "ISA" relationship in the ER model [6]. Thirdly, we want to avoid to define yet another Object-Oriented model that would support more types of relationships [4]. Instead, we are more interested in making the ER approach Object-Oriented [12] and hence, effectively supporting the mapping of ER schema into any Object-Oriented one [11]. At the same time, our approach supports evolution of the current-state-of-the-commercial-art of database systems, that is, relational database systems.

The follow up of this work is to build a prototype for the exploration of schema evolution in multiparadigmatic access of databases. As presented in the beginning of the paper and illustrated by Figure 1, through the graphic user interface and mapping schemes for EVER diagrams, changes to an ER diagram can be made transparent to application programs and interactive users. In other words, the application programs

and users can access all objects in the database using their schemas.

References

[1] M. Ahlsen and et. al. Making Type Changes Transparent. In *Proceedings of IEEE Workshop on Language for Automation*, 1983.

[2] J. Banerjee, W. Kim, H. Kim, and H.F. Korth. 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 international Conference on Extending Database Technology*, Mar., 1992.

[4] S. E. Bratsberg. Unified Class Evolution by Object-Oriented Views. In *Proceedings of the 11th Internaltional Conference on Entity-Relationship Approach*, 1992.

[5] P. Chen. The Entity Relationship Model - Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1), March 1976.

[6] P. Chen. ER vs. OO. In *Proceedings of the 11th Internaltional Conference on Entity-Relationship Approach*, 1992.

[7] S. M. Clamen. Schema Evolution and Integration. *Distributed and Parallel Databases: An International Journal*, 2(1):101–126, January 1994.

[8] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *CACM*, 13(6), 1970.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd edition.* The Benjamin/Cummings Publishing Company, Inc., RedWood City, California, 1992.

[10] H. Kangassalo. Concept D: A Graphical Language for Conceptual Modeling and Data Base use. In *Proceedings of the IEEE Workshop on Visual Languages*, October 1988.

[11] C. T. Liu, P. K. Chrysanthis, and S. K. Chang. Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships. Technical report, TR-94-14, Department of Computer Science, University of Pittsburgh, January 1994.

[12] S. B. Navathe and M. K. Pillalamarri. OOER: Toward Making the E-R Approach Object-Oriented. In *Proceedings of the 8th Internaltional Conference on Entity-Relationship Approach*, 1989.

[13] H. A. Skarra and S. B. Zdonik. Type Evolution in an Object-Oriented Database. In *Research in Object-Oriented Databases* . Addison-Wesley, 1987.

[14] 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), June. 1986.

[15] S. B. Zdonik. Object-Oriented Type Evolution . In *Advances in Database Programming Languages.* Addison-Wesley, 1990.

[16] R. Zicari. A Framework for Schema Updates In an Object-Oriented Database System. In *Proc. of Conference on Data Engineering*, 1991.