JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 9, 657-683 (1993)

*(Invited Paper)*

# Schema Evolution through Changes to ER Diagrams

CHIEN-TSAI LIU, PANOS K. CHRYSANTHIS*, AND SHI-KUO CHANG
*Department of Computer Science*[†]
*University of Pittsburgh, Pittsburgh, PA 15260*
{liuct, panos, chang}@cs.pitt.edu

In order to meet the requirements of new database applications while, at the same time, continue to support existing applications, database systems need to be able to cope with changing database schemas and maintain consistency between instances created under different schemas.

This paper presents an approach to schema evolution through changes to the Entity-Relationship (ER) schema of a database. In order to facilitate changes to the ER schema, we enhanced the basic constructs of ER diagrams with constructs that specify versions of entity and relationship types, and relationships between attributes in different versions. This approach has the advantages of being graphic-oriented and closer to the designer's perception of data rather than to the logical database schema which describes how data are stored in the database. The underlying database structure is re-organized, if necessary, to accommodate new data without changes affecting existing objects. In this way and through the construction of views, modifications of existing programs are avoided while all objects in the database are accessible to all application programs, both new and old.

*Keywords:* Database Schema Evolution, Database Schema derivation, Entity-Relationship model, Relational databases, Change Specification Language, Database consistency.

## 1. INTRODUCTION

As a database is usually employed to capture changes over time, there is a need to be able to reflect these same changes to the database, so that 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 frequently re-organize a large database. Furthermore, it is neither easy nor practical to frequently modify complex application programs such as those found in database systems [16]. Thus, it becomes necessary to continue to support existing application programs providing access to objects created under both the previous and the new database schemas.

Received October 21, 1993; revised March 8, 1994.
Communicated by Wei-Pang Yang.
*This work is partially supported by the National Science Foundation under grant IRI-9210588.
†Phone: (412) 624-8838, (412) 624-8490; FAX: (412) 624-8854.

Various approaches to the problem of changing database schemas and maintaining consistency between instances created under diffeent schemas have been proposed, particularly in the context of Object-Oriented databases [2, 4, 8, 18, 21, 22]. This paper discusses an alternative way to support *schema evolution* based on the Entity-Relationship (ER) approach for data modeling [5, 19]. This is a result of our effort to better understand the semantics of changes. We chose to examine the semantics of changes in the context of the ER model for two reasons: first, because the ER model supports many types of relationships whereas Object-Oriented models primarily support only one type of relationship, similar to the "ISA" relationship in the ER model [6] and second, in order to avoid defining 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 [15] and, hence, able to effectively support the mapping of an ER schema into any object-oriented one. At the same time, our approach supports schema evolution of current state-of-the-commercial-art database systems, that is, relational database systems.

A key idea in the proposed ER approach to schema evolution is the three-level schema mapping. As indicated in Fig. 1, changes on an *ER diagram* representing the structure of entity and relationship types (i.e., database schema) are specified using a high level specification language, called **SPEER** (for SPecification

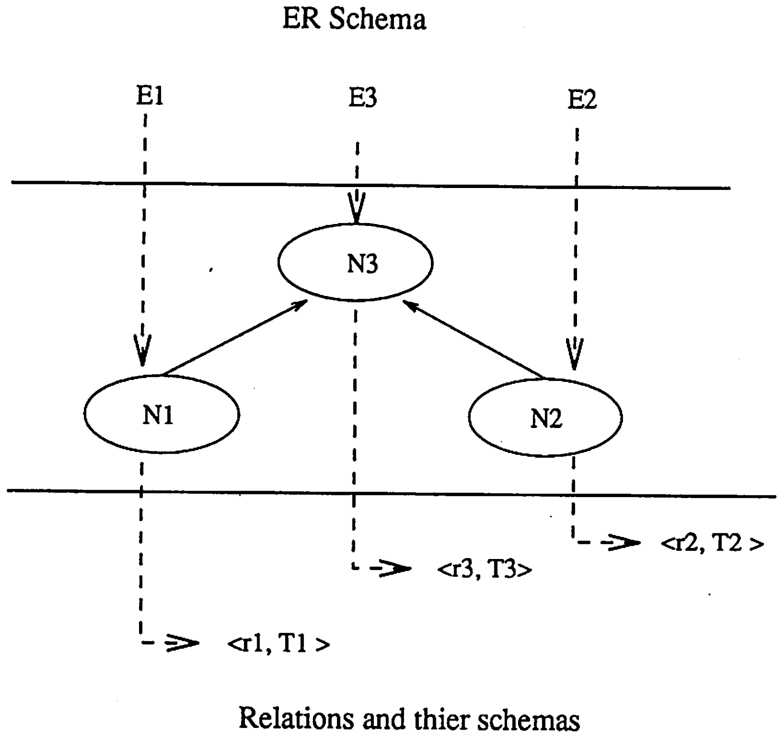


Fig. 1. Three level schema mapping.

of Evolving Entities and Relationships). The specification of the changes on the ER diagram are then mapped to a database implementation independent representation called *version derivation graphs* (VDGs), which is subsequently mapped into the structures of the underlying database [14]. Through this three-level mapping, our approach provides a uniform specification of changes to a database schema independent of the underlying implementation database model.

When the schema of an entity or relationship type (or schema for short) is changed, a new version of the schema is then created. 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, programs are grouped into classes according to the schema version (or view) they they refer to. While the views of schema versions never change, the underlying database structure is re-organized, if necssary, to reflect the evolution of the schema, and thus, the invariance of the programs' interface is preserved. In other words, through the construction of views, modifications of existing programs are avoided while all objects in the database are made accessible to all application programs, both new and old. In this paper, we assume that existing objects retain their representation whereas new data are represented in *complementary* representations and are related to existing objects.

The rest of the paper is organized as follows. In the next section, we survey the various object-oriented approaches to schema evolution. In Section 3, we present the concepts of the ER data model, formally analyze and classify the relationships of attributes between the schemas before and after a change, and discuss issues in maintaining a consistent database manipulated through different schema versions. In section 4, we introduce the SPEER language for expressing the relationships between the new and old schemas, and illustrate the specification of changes to ER diagrams using several examples. Section 5 presents a methodology for mapping the specification of the changes to a schema onto the corresponding VDGs, whereas their transformation into an underlying database, assumed to be relational [9, 10] is discussed in Section 6. The paper concludes with a summary and future work.

## 2. RELATED WORK

Although the ER data model has long been used for conceptual database design, to the best of our knowledge, no work has been reported on schema evolution in the context of the ER model. On the contrary, much of the work on schema evolution has been conducted in the context of object-oriented database systems (OODBS). Generally speaking, approaches to schema evolution can be divided into three categories, namely, *schema modification, schema versioning* and *schema derivation*, based on the *external representation* of the structure of the objects in the database (object schema) presented to application programs and interactive users, and the *internal representation* of the objects used in the underlying database.

*Schema modification* approaches always support a *single schema* and a *single internal* representation for each object [2, 11, 22]. In these approaches, there are a set of rules for performing changes to an object schema and a set of checks for maintaining syntactically correct types and methods across schema versions. In this way, all existing objects can be converted to conform to the new schema. Because of this, the schema modification approach does not support the transparency of change for 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, 18]. One version of an object corresponds to one version of the schema. The instantiation of an object to a schema version is performed at the time of the creation of the object. Objects created under different schema versions must also be accessible through any other schema version. This is achieved through the use of functions that convert the representation of objects from one schema version to another. There are different techniques for the implementation of conversion functions. Ahlsen et al. proposed function composition for making changes to a schema transparent to application programs [1]. The relationships of attributes are specified on two consecutive schema versions. Thus, the objects associated with schema version $V_i$ can be converted into ones with any other schema version, $V_k$, by composing the functions betwen $V_i$ and $V_k$. Skarra and Zdonik proposed another approach to the conversion of objects across schema versions based on *exception handlers*. When a schema versions, $V_i$, is changed, for all other schema versions, $V_j$, a designer is expected to associate exception handlers with each attribute (1) defined in $V_i$ but not defined in $V_j$ or vice versa, and (2) whose type defined in $V_i$ is different from that defined in $V_j$ [18].

In the schema versioning approaches, objects belonging to a version of a schema always stay in that version. Therefore, if a schema associated with the initial schema is subsequently augmented, it will not be possible for the objects associated with the initial schema to be updated by programs associated with a later version without *loss of information*. For example, suppose the domain of an attribute of objects associated with an early schema version is augmented in a later one. If a program associated with the later schema version updates the attribute of an object created by a program associated with the earlier schema version, then there may be insufficient storage for the augmented attribute.

*Schema derivation* approaches support *multiple schemas* for an object and a *common internal* object representation [3, 4, 7, 21]. 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. That is, the objects are presented to the programs as database views on objects in the underlying database. A new schema version is derived from the existing and/or other derived schemas, and may contain less information (*information capacity reducing*), the same amount of information (*information capacity preserving*), or more information (*information capacity augmenting*) than do the old ones. The approach proposed by Tresch and School can support schema evolution which results in *information capacity reducing* and *information capacity preserving* without database reorganization [20]. Zdonik proposed an approach in which an object is represented by multiple views. Each view, i.e., the layer of interfaces, corresponds to a schema version [21]. When a schema version is changed, the objects created under the changed version are augmented, if necessary, by appending extra storage to accommodate the new information, while the view for the new schema version is defined based both on the new properties of the schema version and the properties defined in the view for the previous schema version. Clamen's approach represents an object associated with different schema versions by means of *multifacets* [8, 7]. Each facet of objects corresponds to a version of their schema. An object is represented as a disjoint union of each facet. The relationships among facets is expressed in terms of relationships between attributes viewed from different facets. Clamen classified the attribute relationships into four groups: *shared, derived, independent* and *dependent*. An attribute is *shared* when it is common to both versions. An attribute is *derived* when its value can be derived directly from the attributes in the other version. An attribute is *independent* when its value cannot be affected by, nor affect, other attributes. Finally, an attribute is *dependent* when its value cannot be derived but is affected by other attributes. The attribute relationships can be expressed using functions. Thus, a change to an attribute viewed from a facet can be propagated to the corresponding attribute(s) in another facet by invoking the functions. Hence, object consistency among facets can be maintained.

Bertino proposed a view definition language for schema evolution that provides constructs which allow attributes and methods to be added to a view corresponding to a new schema [3]. In this approach, views are organized as a view derivation hierarchy which is different from the class hierarchy. Hence, a view (or a new schema) and its base classes are not directly related. Along similar lines, Bratsberg adopted the approach of separation of a database into two parts, namely, *intent* and *extent* [4]. Intent represents the derivation hierarchy among schema versions whereas extent represents the associations of objects with schema versions. Since the organization of intent hierarchy is based on the order of the creation of classes, this approach seems to introduce another data model which is different from the Object-Oriented model in structuring superclasses and subclasses. Although the existing derivation approaches presented above 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 and, as such, supports multiple schema versions and a common internal object representation, referred to as the *complete object*. However, by allowing linear schema evolution, i.e., by allowing only the most recent schema to evolve, and by considering eliminated and resumed attributes in the evolution history, our approach effectively resolves the inconsistency problems along multiple derivation paths mentioned above.

## 3. EVOLUTION IN ENTITY-RELATIONSHIP DIAGRAMS

In this section, after reviewing the basic ER model, we discuss certain possible changes to the ER diagram. We formally analyze the attribute relationships between two schema versions, and we explicitly express these relationships in terms of functions. Finally, we present rules for maintenance of a consistent database in the context of the ER model.

## 3.1 The Entity-Relationship Model

The basic ER data model supports two semantic primitives, *entities* and *relationships*, between entities, in terms of which the structure of a database, i.e., the database schema, is described. An *entity type* is a set of entities having the same properties or attributes. Similarly, a set of relationships among entities from a number of entity types form a *relationship type* among these entity types. The *degree* of a relationship indicates the number of entity types that participate in the relationship whereas the *cardinality* of a relationship specifies the mapping of the associated entity occurrences in the relationship.

Attributes are used to describe an entity or a relationship. Each attribute is associated with a domain that defines the possible values for the attribute. A domain can be defined as a pair of *type* and *range* of values, denoted by *type*[*range*]. The type can be any system supported data type such as *integer, real* or *string. Range* is a constraint on a type. For example, in the case of an integer type, the *range* can be the smallest and largest ineger values, represented by *integer*[*lower* .. *upper*]. The range associated with a *char* type can be the set of allowable characters, denoted by *char*[$char_1$ .. $char_2$], and the range of a *string* type can be represented by *string*[m], where *m* is the length of the string.

Each entity usually has an *identifying* or *key* attribute, whose value is distinct from any other entity in the same entity type. The instances of a relationship type can be identified by the key attribute of the entities involved in the relationship and are called *foreign attributes* to the relationship.

An *ER diagram* is a graphical representaiton of a database schema consisting of *rectangular nodes* representing entity types and *diamond-shaped nodes* connected to rectangular nodes representing relationship types. The number of connections of a relationship type to entity types indicates the degree of the relationship type. The attributes of entity or relationship types are represented by circles. The identifying attribute is indicated by a filled-in circle. Fig. 2 shows the relationship Madeby between entities Car and Maker using an ER diagram. For simplicity of presentation, cardinalities of relationships are not considered here.

In what follows, we will use *object* to refer to an instance of an entity type or a relationship type, and *object schema* or *schema* to refer to the description of an entity type or the relationship type when they need not be distinguished. Hence, an ER diagram (i.e., the ER database schema) consists of a set of schemas which define the interface for application programs to create, update and access objects in the database.

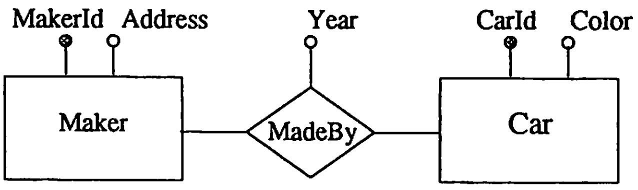


Fig. 2. A partial ER diagram for modeling cars.

## 3.2 Structurally Consistent Changes to ER Diagrams

An ER diagram is structurally correct (or well formed) if and only if

- Each entity or relationship type is associated with a unique name.
- Each attribute of an entity or a relationship type is associated with a unique name and a domain. Attributes of different entity or relationship types may be associates with the same name.
- Each entity type has an identifying attribute.
- Each connection (edge) connects an entity type to a relationship type. That is, in an ER diagram, there are no dangling edges or edges connecting two entity types or two relationship types.
- Each relationship type is connected to *at least* two entity types. That is, the number of edges associated with a relationship type is equal to or greater than two.

We expect that the initial ER diagram, which is the one created at the time the database was first designed, is structurally correct. When a change to an ER diagram is made, we require that the resulting diagram must also be structurally correct. We considered the following possibilities of change.

1. A node and its associated edges may be added to an ER diagram, corresponding to the introduction of a new entity type or a new relationship type.
2. A node and its associated edges may be removed from an ER diagram, corresponding to the dropping of an existing entity type or relationship type.
3. Edges may be added to, or removed from, an ER diagram, reflecting changes in the relationships among the existing entity types in the database.
4. A node may be split into several nodes, and a number of nodes may be merged into a single node as a result of aggregation and decomposition of entity types, respectively.
5. Attributes may be added to, or dropped from, nodes, corresponding to the gain and loss of attributes because of new properties of the objects and requirements of the application programs.
6. The domain of attributes may be changed, expanded or reduced over time.

A close examination of the above possibilities shows that changes may involve exisitng attributes. For example, when the attributes of a schema are changed, the attributes of the new version of the schema may be common or derived from the attributes of the old version of the schema. In other words, there exist relationships of attributes between the new and old schemas. These relationships reflect the semantics of changes on the old schema that result in the new schema. In order to facilitate the maintenance of object consistency across the different schema versions and efficient storage allocation, in the following section, we perform a detailed analysis of the relationships among the attributes of two schema versions.

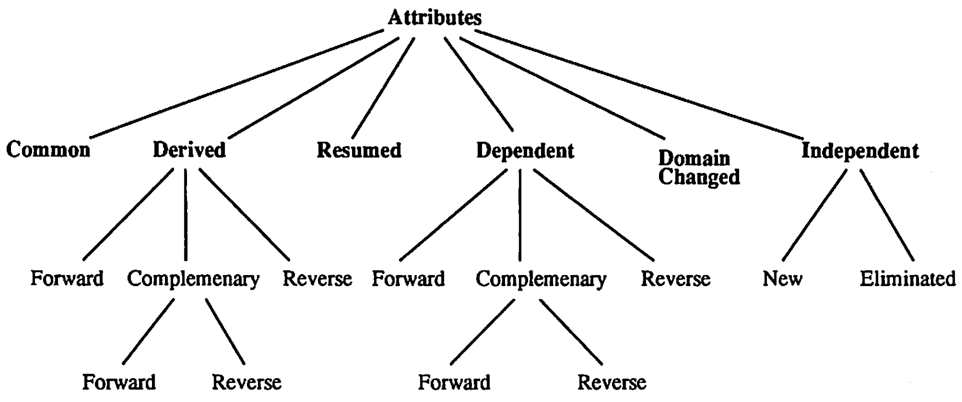


Fig. 3. The classification of attributes.

## 3.3 Classification of Attribute Relationships

When a schema evolves, the relationships between the attributes of the old and new schemas capture the semantics of installed changes. These relationships provide crucial information for maintaining object consistency and reorganization of the objects in the underlying database. For this reason, given two schema versions, we classify the attributes of the two schemas based on the relationships of their names, their values, and their domains. This classification refines the one proposed in [7] (see Fig. 3).

- **Common** attributes: An attribute is said to be *common* to the two shcemas if the names and domains of the attribute in the two schemas are identical.
- **Domain-changed** attributes: An attribute is said to be **domain-changed** if the names of the attribute in the two schemas are exactly the same but its domain is different.
- **Renamed** attributes: An attribute is said to be **renamed** if the attributes in the two schemas have different names but exactly the same domains.
- **Resumed** attributes: An attribute is said to be **resumed** if the attribute was deleted from an earlier schema version but is added back to a later 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. Let $B$ be an attribute derived from attributes $\{A_1, A_2, ..., A_k\}$. The relationship can be represented by $B = f(\{A_1, A_2, ..., A_k\})$.
- **Dependent** attributes: An attribute, say $B$, is said to be **dependent** if the value of the attribute is affected by changes to the values of other attributes,

say $\{A_1, A_2, ..., A_k\}$, but the value of the dependent attribute cannot be *derived* from the values of the same attributes $\{A_1, A_2, ..., A_k\}$.

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

The relationships of each group of attributes can be expressed by using a general form of functions. There are four kinds of functions used in our framework.

**Identity function.** Let $a$ and $b$ be attributes. If $a$ and $b$ are identical, their relationship can be represented using an *identity function (I)* such that

$$a = I(b), \text{ or simply, } a \equiv b.$$

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

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

**Default function.** Let $a$ be an attribute. If the value of $a$ of an object is unspecified, but the value is required by a program, then the value can be acquired by using a *default function* (*default*). The value is the so-called *default value*. By assigning a default value to an unspecified attribute in a previous schema, the needs of application programs associated with different schema versions can be satisfied.

The specification of the attribute relaitonships between two consecutive schema versions can be expressed in terms of these four kinds of functions. In order to indicate the mapping direction of a function, functions are specified as *forward* or *reverse: forward* indicates that the mapping is from the old to the new schema version, and *reverse* indicates that the mapping is from the new to the old schema version.

Attribute relationships may exist between schema versions $V_i$ and $V_l$, $i < l$. For example, an attribute of $V_i$ can be resumed in another $V_l$ if there is no such attribute in any schema version $V_j$ in between $V_i$ and $V_l$. The resumed attribute allows for capturing these types of relationships as relationships of two consecutive schema versions.

The attribute relationships are mainly used to maintain consistent objects in the database with respect to observers who view the database from different schema versions. In the following sections, we present criteria for maintenance of data consistency across schema versions and introduce the language for specifying changes to ER diagrams.

## 3.4 Maintenance of Object 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.** 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. We assume, here, that the semantics of the functions and the procedures associated with the functions must be correct in the sense that their results must meet the users' expectations, and satisfy the integrity constraints.

**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 based on 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 a key attribute: *the mapping of 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. For example, consider the database schema in Fig. 2. Suppose that schema Car evolves

Table 1. **The addition of attribute Fuel to schema Car.**

| The old schema | The new schema |
|---|---|
| *CarId: string*[20] | *CarId: string*[20] |
| *Color: string*[12] | *Color: string*[12] |
| | *Fuel: integer*[1..2] |

due to the addition of attribute Fuel as shown in Table 1. Fuel's possible value is either *leaded* or *unleaded*. The cars belonging to the old schema version burn leaded gas only. The cars belonging to the new schema version burn either leaded or unleaded gas.

Assuming that the measurements for inspection of unleaded cars are different from the leaded cars, if a program associated with the old schema version is used to analyze car inspection records, then it should only access the cars that burn leaded gas. Since all the cars associated with the old schema version burn leaded gas, the semantics of the database is clear to the programs associated with that old schema version. They view the database as a set of *leaded cars*. However, with the introduction of unleaded cars, the semantics of the evolved database is no longer the same. In our framework, we provide facilities to allow a designer to specify the conditions under which programs retain a consistent view between the previous and the evolved databases.

In summary, the rules for maintenance of a consistent database for application programs are as follows. The objects created under both schemas are *consistent* with respect to the application programs associated with a schema version if:

1. Each attribute, irrespective of whether it is of the new or old schema, must be classified into a group and associated with functions in accordance with its group. The functions are used to maintain object consistency between the two consecutive schema versions.
2. Let *Old*(*key*) and *New*(key) be key attributes of the old and new schemas, respectively. Their mapping must be one-to-one. That is, $New(key) = f(Old(key), \Phi)$ AND $Old(key) = f'(New(key), \Phi)$, where $\Phi$ represents additional information or empty.
3. Let *Old*(*DB*) and *New*(*DB*) be the databases corresponding to the old and new schema, respectively. Let $g$ be a function representing the view of programs associated with a schema version. Then, the invariant view for these programs can be expressed as:

   $$g(Old(DB)) = g'(New(DB)),$$

   where $g'$ is a function used to construct the view through which the programs can preserve the same view to the database after evolution.

## 4. SPECIFICATIONS OF CHANGES TO ER DIAGRAMS

### 4.1 A Language for Database Schema Evolution

In order to facilitate changes to ER diagrams, we have designed a language for specification of changes to the ER diagrams called **SPEER** (SPecification of Evolving Entities and Relationships). The language extends the constructs of ER diagrams to allow a designer to specify the following relationships:

- the derivation path of a 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 ER diagrams);
- the invariant views of programs to the database.

The derivation path indicates from where the new schema evolves. The attribute relationships specify the effect of changes to an attribute on the others and can be expressed using functions. A change to an edge between an entity and a relationship type implies that the participation of the entity type in the relationship type is either established or dropped. Consequently, the relationship type needs to be developed by 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. We summarize the change specification language for evolving entities and relationships (in ER diagrams) in Fig. 4. The bold-typed words are terminals, and the italic words are nonterminals. We will briefly describe each of the constructs below.

The derivation path of the new schema to the old schemas is specified using the constrct

**EVOLVE SCHEMA** ⟨*Old*(*OldSchemas*)⟩ **INTO** ⟨*New*(*NewSchema*)⟩.

Old(OldSchemas) is a set of schemas from where the new schema (New (NewSchema)) evolves. The schema version OldSchemas includes all the attributes of the old schema version and the eliminated attributes in the previous schema versions. Functions *Old*() and *New*() indicate the sets of old or new versions of a schema. The attributes of the new schema are listed in the AttributeList.

The specification of attribute relationships consists of two parts: the classification group (GroupSpec) and function specification (FunctionSpec). The classificaiton group consists of the definition (AttributeDef) and the group (GroupDef) of an attribute. The group of an attribute, irrespective of whether it is of the new or old schema, must be specified. However, only the definitions of those attributes in the new schema which do not exist in the old schemas need to be specified. The relationships of one attribute to the others can be expressed in a form of functions presented in Section 3.3. The specification of a function consists of two parts: the function definition and its implementation. The mapping direction is not explicitly specified in a function specification. The direction can

```
EVOLVE SCHEMA ⟨Old(OldSchemas)⟩ INTO ⟨New(NewSchema)⟩ AS
    ATTRIBUTES {⟨AttributeList⟩}
    CLASSIFICATION {⟨GroupSpec⟩}+
    FUNCTIONS {⟨FunctionSpec⟩}+
    [INVARIANT VIEWS {(⟨AccessSpec⟩)}+]
    [ADD | DEFUNCT EDGES {⟨EdgeSpec⟩}+]
    ⟨GroupSpec⟩ ::= {(⟨AttributeDef⟩ WITH ⟨GroupDef⟩); }
⟨AttributeDef⟩ ::= [⟨AttributeName⟩ | ⟨AttributeName⟩: ⟨type⟩[⟨rang⟩]]
    ⟨GroupDef⟩ ::= [⟨Group⟩ | ⟨Group⟩ [TO | FROM]{⟨OldSchemas⟩}]
    ⟨Group⟩ ::= [COMMON | DOMAIN-CHANGED |
            FORWARD-DERIVED | REVERSE-DERIVED |
            FORWARD-COMPLEMENTARY-DERIVED |
            REVERSE-COMPLEMENTARY-DERIVED |
            FORWARD-DEPENDENT | REVERSE-DEPENDENT |
            FORWARD-COMPLEMENTARY-DEPENDENT |
            REVERSE-COMPLEMENTARY-DEPENDENT |
            NEW | ELIMINATED | RESUMED]

    ⟨FunctionSpec⟩ ::= ⟨FunctionDef⟩
        [WITH IMPLEMENTATION { ⟨FunctionBody⟩) }]
    ⟨FunctionDef⟩ ::= (⟨AttributeName⟩=⟨FunctionName⟩(⟨ArgumentList⟩));
    ⟨ArgumentList⟩ ::= [⟨AttributeName⟩ | ⟨ProcName⟩]
    ⟨EdgeSpec⟩ ::= (BETWEEN ⟨EntityType⟩ AND ⟨RelationshipType⟩
                    ⟨ParticipationSpec⟩);
    ⟨ParticipationSpec⟩ ::= PARTICIPATION [before | after]: ⟨FunctionSpec⟩
    ⟨AccessSpec⟩ ::=  {⟨SchemaName⟩ ACCESS WITH CONDITIONS
            ⟨ConditionId⟩: ⟨Condition⟩); }
```

Fig. 4. The specification of evolving ER diagrams.

be easily inferred from the schema version to which the attributes defined in the co-domain and domain of the function belong. The addition or removal of an edge between an entity and a relationship type can be expressed in the specification of an edge (EdgeSpec). This change also leads the changes to the foreign attributes of a relationship type. The conditions for maintenance of invariant program views to the evolved database can be expressed in the access specification (AccessSpec). The access specification consists of a set of assertions that objects in the database

**Table 2. The augmentation of the domain of an attribute.**

| The old schema | The new schema |
|---|---|
| *CarId: string*[20] | *CarId: string*[20] |
| *Color: string*[12] | *Color: string*[12] |
| *MPG: integer*[0 .. *Max*(*short*)] | *MPG: integer*[0 .. *Max*(*long*)] |

must satisfy. The key attribute of the new schema must also be specified if it is different from the old one.

In the following sections, we will present examples using the SPEER language to specify changes to ER diagrams. We assume that all the changes must satisfy the constraints in maintaining structurally consistent ER diagrams and consistent databases.

## 4.2 The Changes to the Domain of an Attribute

Let $dom_{old}$ and $dom_{new}$ be the old and new domains, respectively. There are four cases of changes to the domain of an attribute: the domain is strictly augmented (i.e., $dom_{old} \subset dom_{new}$), the domain is strictly reduced (i.e., $dom_{new} \subset dom_{old}$), the domain is overlapping (i.e., $dom_{new} \cap dom_{old} \neq \phi$), and the domain is disjoint (i.e., $dom_{new} \cap dom_{old} = \phi$). The mapping between the new and old domain is expected to be specified using the proposed language. Let us use the example shown in Fig. 2 to illustrate the augmentation of the domain of attribute MPG, mileage per gallon, of schema Car from a short integer to a long integer. The range of a short (long) integer is represented [0 .. max(short)] ([0 .. max(long)]). All the other attributes remain unchanged. The attributes of the new and old schemas are shown in Table 2.

Assuming that function $f_{v1}$ maps the domain of attribute MPG in the new schema (New(MPG)) to that of attribute MPG in the old schema (Old(MPG)), and that function $f_{v2}$ maps the domain of attribute Old(MPG) to that of New(MPG), then the specification of the change can be expressed in SPEER as:

```
EVOLVE SCHEMA Old(Car) INTO New(Car) AS
    ATTRIBUTES {MPG, Color, CarId};
    CLASSIFICATION {
      {Color, CarId} WITH COMMON;
      {MPG: integer[0 .. max(long)]} WITH DOMAIN-CHANGED};
    FUNCTIONS {
      (New(MPG) = f_v2(Old(MPG);
         WITH IMPLEMENTATION New(MPG) = Old(MPG));
      (Old(MPG) = f_v1(New(MPG);
         WITH IMPLEMENTATION
           if New(MPG) ≥ max(short_integer)
           then Old(MPG) = max(short_integer) else Old(MPG) = New(MPG))}.
```

Since the old schema cannot be seen by the new programs nor can it be used for future evolution, we consider it as a *defunct* schema. The edges connecting the old schema and other schemas are also defunct, and are redirected to the new schema. The defunct schemas and the new schema are mapped into VDGs for storage allocation and construction of views for each schema version.

### 4.3 Addition or Deletion of Non-Key Attributes

Most of the changes to a relational database schema are either additions or deletions of non-key attributes [17]. This is also expected to be the case for any other type of database schema. This section presents how the proposed language can be used to express the relationships for addition to, or removal of non-key attributes from a corresponding ER schema.

Consider again the evolution of schema Car due to the addition of attribute Fuel with data values *leaded* and *unleaded*, discussed in the previous section (see Table 1). Since Fuel cannot be derived from or affected by any of the existing attributes, it is a new, independent, attribute. The objects associated with the old schema correspond to cars that burn leaded gas. This fact can be expressed by a *default* function for conversion of objects in the old database to ones in the new database. Because the new database consists of cars that burn leaded or unleaded gas, this may introduce inconsistent interpretation of the database for the programs associated with the old schema. Hence, the designer is expected to express the access conditions of these programs to the new database. Let values *leaded* and *unleaded* be represented by integers 1 and 2, respectively. The change can be expressed as follows:

```
EVOLVE SCHEMA Old(Car) INTO New(Car) AS
    ATTRIBUTES {CarId, Color, MPG, Fuel};
    CLASSIFICATION {
      ({CarId, Color, MPG} WITH COMMON);
      ({Fuel: integer[1..2]} WITH NEW)}
    FUNCTIONS {
      (Fuel = default) WITH IMPLEMENTATION (Fuel = 1)};
    INVARIANT VIEWS {
      (Old(Car) ACCESS WITH CONDITIONS Ac1: (Fuel = 1))}.
```

Similarly to the addition of a non-key attribute to a schema, when a non-key attribute is deleted, the default value of the attributes for the objects associated with the new schema is expected to be specified. Thus, the programs associated with the old schema can correctly access the evolved database.

### 4.4 Replacement of a Key Attribute of an Entity Type

Since the values of a key attribute are used to uniquely identify objects associated with an entity or a relationship type, our approach requires that changes to a key attribute must satisfy the *key consistency* rule (see Section 3.4), which requires that the mapping between the old and new keys must be one-to-one.

If the key attribute is a *foreign attribute* of a relationship type, then the affected relationship type must evolve, too. Let us use the example shown in Fig. 2 to demonstrate how a change to a key attribute can be specified in SPEER. Suppose the key attribute CarId of entity type Car is replaced by a new attribute VIN. The mapping between CarId and VIN must be a one-to-one correspondence; otherwise, the change must be rejected. The mapping can be implemented by a table lookup. That is, given a value of CarId, we can find a unique value of VIN, and vice versa. Thus, attributes CarId and VIN are dependent on each other. Let CarToVeh and VehToCar be tables for mapping from the values of CarId to VIN, and from the values of VIN and CarId, respectively. This conversion can be implemented as a selection ($\sigma$) on table CarToVeh (or VehToCar) based on a given CarId (or VIN), and then followed by a projection ($\pi$) on the selected cars based on attribute VIN (or CarId). The specification of the change to the key attribute of schema Car can be expressed in SPEER as follows:

```
EVOLVE SCHEMA Old(Car) INTO New(Car) AS
    ATTRIBUTES {Color, VIN};
    CLASSIFICATION {
      ({Color} WITH COMMON);
      ({CarId} WITH DEPENDENT);
      ({VIN: string[20]} WITH DEPENDENT)};
    FUNCTIONS {
      ($VIN = f_{v12}(CarId, CarToVeh)$)
        WITH IMPLEMENTATION
          $VIN = \Pi_{VIN}(\sigma_{(VIN = CarId)}(CarToVeh)))$;
      ($CarId = f_{v21}(VIN, VehToCar)$)
        WITH IMPLEMENTATION
          $CarId = \Pi_{CarId}(\sigma_{VIN = CarId}(VehToCar)))$};
    KEY ATTRIBUTE VIN;
```

The relationship type MadeBy must also evolve because the key attribute of Car is a foreign attribute of MadeBy. The attribute relationships needed for evolution of the relationship type MadeBy can be obtained from the specification of the change to schema Car. This propagation of change can be handled in a way similar to the above.

## 4.5 Addition or Deletion of Schemas and Edges

When adding a new schema to an ER diagram, the attributes of the schema must be defined, and the edges for establishing relationships between the new schema and the existing schemas must be specified. The addition of an edge between an entity type and a relationship type implies that the relationship type gains a foreign attribute which is the key attribute of the entity type. Similar to the case of addition of an attribute to a schema, the default value of the foreign attribute must be specified. That is, there should exist an implicit participation of the entity type in the relationship type before the change. For example (refer to Fig. 2), suppose a new entity type, named Dealer, and an edge between Dealer

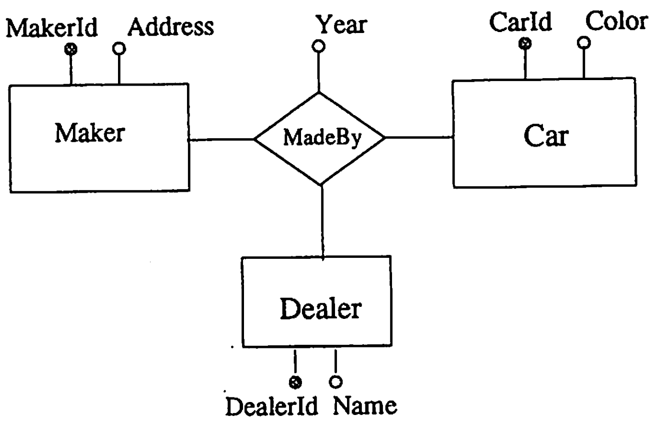


Fig. 5. The addition of a node and an edge to a relationship type.

and relationship type MadeBy are to be added to the ER diagram. Although there is no entity type Dealer shown in the ER diagram before the change, the implicit participation of a dealer into relationship type MadeBy may be through the maker who is a dealer. Thus, the specification of the addition of an edge between Dealer and MadeBy can be expressed as follows. The specification of Dealer is not shown here because a new schema can be created using the constructs of ER diagrams. The resultant ER diagram is shown in Fig. 5.

```
EVOLVE SCHEMA Old(MadeBy) INTO New(MadeBy) AS
    ATTRIBUTES {MakerId, DealerId, Year};
    CLASSIFICATION {
      ({DealerId: string[20]} WITH NEW)};
    ADD EDGES {(BETWEEN Dealer AND MadeBy;
      PARTICIPATION before: DealerId = default
        WITH IMPLEMENTATION
          DealerId(x) = MakerId(x))};
    KEY ATTRIBUTE {DealerId, MakerId}
```

When an edge between an entity type and a relationship type is deleted, the role of the entity type participating in the relationship type may become implicit. Similarly, the implicit participation can also be described using SPEER. In the next section, we present the specification for schema merging and schema splitting.

## 4.6 Merging and Splitting Entity Types

The above examples demonstrate the evolution of a single schema. In practice, a schema is often derived by merging two or more entity types, or by splitting one entity type into several ones. Without loss of generality, we require that the key attributes of the to-be-merged schemas be exactly the same. If the key attributes are different, the designer can rename the key attributes with a common

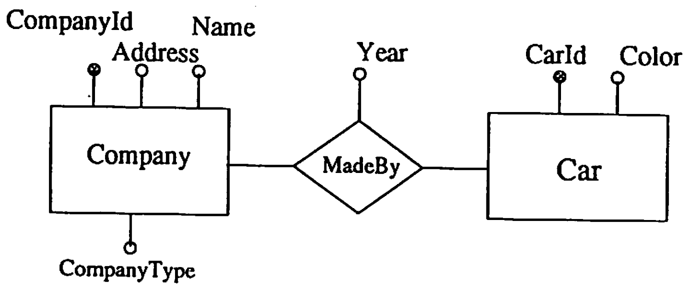


Fig. 6. Merging two entity types Dealer and Maker into Company.

name, and schema merging can then proceed. Schema splitting can also be handled in way similar to schema merging.

Consider the ER diagram in Fig. 5. Suppose that a new entity type, Company, is created by merging entity types Maker and Dealer (see Fig. 5 and 6). Since their key attributes are different, they must be renamed with a common attribute name, say CompanyId. The new entity type, Company, inherits attributes {CompanyId, Name, Address} and owns an additional attribute, Company Type, indicating the type of company. Attribute CompanyType is new to the schemas Maker and Dealer. The default value for the attribute of the existing objects depends on where the objects belong, and can be expressed as follows:

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

Function $schema_{type}(x)$ returns the name of the schema to which object $x$ belongs. Thus, programs that refer to the old schema may still access the appropriate objects in the database described by the merged schema. For example, the programs that refer to entity type Maker may only need to access the objects whose value of CompanyType is equal to *maker*. The specification of merging the two schemas Dealer and Maker is shown below.

```
EVOLVE SCHEMA Old({Dealer, Maker}) INTO New(Company) AS
    ATTRIBUTES {CompanyId, CompanyType, Name, Address};
    CLASSIFICATION {
      ({CompanyId} WITH COMMON TO {Dealer, Maker});
      ({Name} WITH COMMON TO {Dealer});
      ({Address} WITH COMMON TO {Maker});
      ({CompanyType: string[20]} WITH NEW TO {Dealer, Maker}}
    FUNCTIONS {
      ((CompanyType(x) = default)
        WITH IMPLEMENTATION
          (if Schema_type(x) = Dealer then CompanyType(x) = dealer
          else CompanyType(x) = maker))};
```

```
KEY ATTRIBUTE {CompanyId};
ADD EDGES {(BETWEEN Company AND MadeBy)}
INVARIANT VIEWS {
  (Maker ACCESS WITH CONDITIONS
    (ac1: CompanyType = maker));
  (Dealer ACCESS WITH CONDITIONS
    (ac2: CompanyType = dealer))};
```

## 5. A CONCEPTUAL MODEL FOR OBJECT REPRESENTATION

In order to support different implementation database models, instead of translating the change specification presented in the previous section directly into an underlying database model, our approach transforms it into a conceptual representation called the *version derivation graph* (VDG), which is subsequently mapped into an underlying database model.

A VDG consists of a set of nodes and a set of directed edges. Each node corresponds to a schema version recording the attributes of the schema, the relationships of the attributes of the schema to the attributes of the previous and following ones, and the conditions for maintaining the invariance of program views. When a new schema version is specified in SPEER, a new node representing the new schema version is added in the corresponding VDG. The storage requirements for the schema version can also be captured in the corresponding VDG node. A directed edge represents the derivation path from schema version Old(Schema) to New(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 storage with attributes of the old schema. Let $E_n$ be a schema version which is derived from schema versions $E_1, E_2, \ldots, E_m$, 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:

- $group(a_i) \in \{new, forward\text{-}dependent, forward\text{-}complementry\text{-}dependent\}$
- $\exists a_k \in E_j \wedge j \in \{1, \ldots, m\}$, such that $a_i = domain\text{-}changed(a_k) \wedge (dom\_size(a_k) \subset dom\_size(a_i))$,

where *domain-changed*($a_k$) returns the attributes that are derived from attribute $a$ and whose domain has been changed; *dom_size*($a$) computes the storage requrements for an attribute $a$.

Since a VDG is 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 base attributes of all the versions of the schema (or the *complete schema*). The object is called a *complete object* of the VDG. Let $B_i$ be a set of base attributes of schema versions $E_i$, $i \in \{1..n\}$. The *complete schema* of schemas $\{E_1, E_2, \ldots, E_n\}$ ($S_c$) can be expressed as: $S_c = \cup_{i=1}^{n} B_i$. In order to illustrate the representation of an object in a VDG, let us consider again the change to the domain of an attribute specified in Section 4.2. As indicated in Table 2, the old schema, Old(Car), consists of attributes CarId, Color and Old(MPG), and the new schema consists of CarId, Color and New(MPG). The New(MPG) has a larger domain than does Old(MPG). Since attributes Color and CarId are common to both schema versions, the new schema can share storage for those attributes with the old schema. On the other hand, attribute New(MPG) cannot share storage with attribute Old(MPG) because the former has a larger domain than does the latter. Thus, the set of base attributes of schema versions New(Car) and Old(Car) is {CarId, Color, Old(MPG), New(MPG)}. This attribute set is also the complete schema of the VDG representing the car entity.

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.

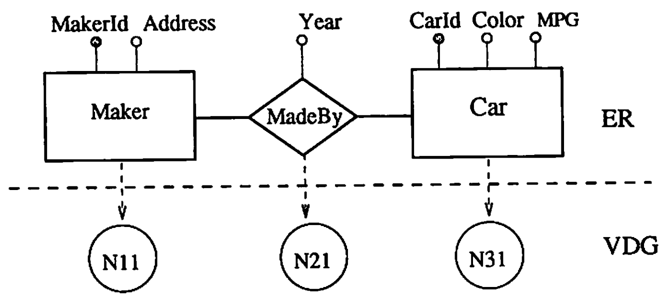
Formally, let schema $E_n$ evolve from schemas $E_1, E_2, \ldots, E_m$ in an ER diagram, and let $N_n, N_1, N_2, \ldots, N_m$ be the nodes in a VDG corresponding to $E_n, E_1, E_2, \ldots, E_m$, respectively. Let $B_n$ be a set of base attributes of schema $E_n$.

```
IF $\exists a_i \in E_n \wedge a_i \in B_n$
then $N_n$ is a non-virtual node,
else $N_n$ is a virtual node.
```

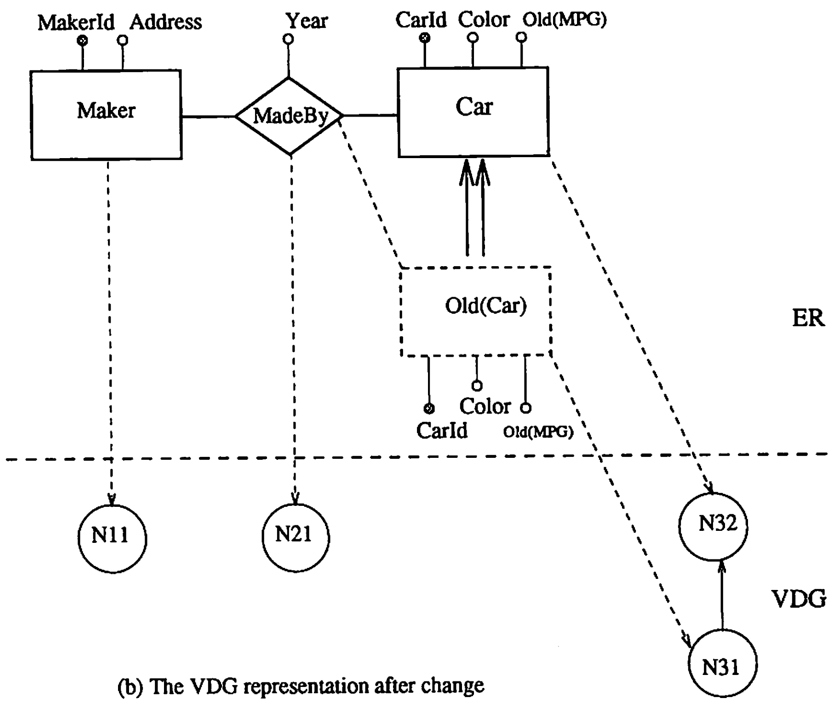
Objects created under a new schema version that maps onto a non-virtual node cannot be stored in the underlying database described by the old schema versions. In other words, the underlying database needs to be re-organized. On the other hand, the objects created from a schema that map onto virtual nodes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

Let us use an example to show the construction of a VDG when a schema in an ER diagram is changed. Consider once again the evolution of schema Car presented in Section 4.2 (Fig. 7). Initially, each schema is represented by one VDG consisting of only one single node representing the initial schema as shown in Fig. 7(a). For example, node $N_{31}$ corresponds to the initial schema Car. Since it is an initial schema, node $N_{31}$ is a non-virtual node (solid circle*). After the domain of the attribute MPG is augmented, the VDG representing schema Car can be reconstructed as follows. First, node $N_{32}$ corresponding to the new schema version is created. Because attribute New(MPG) is a base attribute defined in the new schema, $N_{32}$ is a non-virtual node. Second, the derivation path between schema versions Old(Car) and New(Car) (denoted by the parallel directed line

*Note that Fig. 7 does not contain any virtual node but if it did, the virtual node would have been represented by a "dotted circle."



(a) The VDG representation before change



(b) The VDG representation after change

Fig. 7. The representation of an ER diagram and its VDGs.

shown in Fig. 7(b)) is represented using a directed edge which goes from node $N_{31}$ corresponding to Old(Car) to $N_{32}$ corresponding to New(Car) in the VDG. Finally, the attribute relationships between the new and old schema versions and the invariant of program views to the database are associated with nodes $N_1$ and $N_2$. The resultant VDG for schema Car is shown in Fig. 7(b). The derivation path among schema versions specified in a specification allows the corresponding VDG to be constructed incrementally. The algorithm for construction of a VDG is described as follows:

**Algorithm** Construction of the VDG of a schema.
***Input:***

(i.) A set of ER schema versions $\{E_i | 1 \leq i \leq m\}$ and the specification describes a schema version $E_n$ which evolves from $\{E_i | 1 \leq i \leq m\}$, where $n \notin \}1..m\}$.

(ii.) A VDG consists of a set of nodes $\{N_i | 1 \leq i \leq m\}$. Each node $N_i$ represents a schema version $E_i$, $1 \leq i \leq m$.

***Output:***

A VDG representing the ER diagram after schema evolution.
That is, the VDG includes the new schema version $E_n$ and the attribute relationships between $E_n$ and $E_1 \ldots E_m$.

**Procedures:**

```
1. Create a node for schema $E_n$, say $N_n$, in the VDG, based on the definition
   of base attributes.
2. Associate the definitions of attributes of schema $E_n$ with node $N_n$.
3. For each attribute $b$ of $E_n$
       associate the forward, forward complementary or default function of the
       attribute in the specification with node $N_n$.
4. For each attribute $b_i \in E_j$, $1 \leq j \leq m$
       associate the reverse, reverse complementary or default function of the
       attribute in the specification with node $N_i$.
5. For each schema $E_i$, $i \in \{1..m,n\}$
       associate the conditions of $E_i$ for maintenance of invariant views for
       programs with node $N_i$.
6. For each schema $E_i$, $i \in \{1..m\}$ from which $E_n$ evolves
       connect a derivation edge which goes from $N_i$ to $N_n$.
```

The representation of changes to an ER diagram using VDGs provides independence from the underlying database model. A VDG can be mapped into any implementation database model, for example, a relational, network, hierarchical, or an object-oriented database. In the next section, we will demonstrate the transformation of a VDG into a relational database model.

## 6. MAPPING A VDG INTO RELATIONAL DATABASES

In this section, we present an algorithm that maps a VDG into a relational database schema and constructs views corresponding to different schema versions. 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 of a VDG into the relational database, let us use for a last time the example discussed in Section 4.6, in which two schemas, Maker and

Dealer, are merged together in the single schema Company. The schema Maker and Dealer are initially mapped into corresponding VDGs containing a single non-virtual node, say $N_1$ and $N_2$, respectively. After the two schema are merged, the new schema Company is also mapped into a non-virtual VDG node, say $N_3$, because in addition to attributes CompanyId, Name and Address, it owns the base attribute CompanyType. Being a non-virtual node, $N_1$ is mapped into a relation, $r_1$, whose schema, $T_1$, contains all the attributes of Maker plus one extra attribute, the *object identifier* (Oid): $T_1$(ComanyId, Address, Oid). Similarly, the VDG node $N_2$ corresponding to Dealer is mapped into a relation, $r_2$, with schema $T_2$(CompanyId, Name, Oid). After the schema evolution, $N_3$ requires a new relation to store the base attribute CompanyType. Hence, $N_3$ is mapped into a new relation, $r_3$, with schema $T_3$({CompanyType, Oid}). Thus, objects created under schema Company are fragmented and stored in relations $r_1$, $r_2$, and $r_3$. The complete schema ($S_c$) of the VDG with nodes $N_1$, $N_2$ and $N_3$ is the union of the base attributes of Maker, Dealer and Company: $S_c$ = {CompanyId, Name, Address, CompanyType}.

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 basically defined as a selection of complete objects based on the access conditions associated with $S_i$, followed by 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 using SPEER. Let us illustrate step by step the construction of the views for schemas Maker, Dealer and Company.

*Step 1:* Determine the complete schema of the VDG. As indicated above, the complete schema ($S_c$) of schema Car is $S_c$ = {CompanyId, 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 an initial schema and 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 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 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 $r_1$. The selection condition for identification of a set of the objects created under a schema version can be derived as shown in the following table.

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

*Step 4:* Expand the objects created under a schema version to the complete objects, and then screen the objects that cannot satisfy the specified *conditions* out from the view of the programs. As indicated in the specification, the programs that refer to entity types Maker and Dealer can access the objects with their attribute value *CompanyType = Maker* and *CompanyType = Dealer*, respectively. 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 follows:

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

where $\Pi$ stands for projection, $\sigma$ 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

$$View_{Maker} = \Pi_{(CompanyId,\ Address)}(\sigma_{(CompanyType = Maker)}(\cup_{i=1}^{i=3} Expand(O_i^*)))$$
$$View_{Dealer} = \Pi_{(CompanyId,\ Name)}(\sigma_{(CompanyType = Dealer)}(\cup_{i=1}^{i=3} Expand(O_i^*)))$$
$$View_{Company} = \Pi_{(CompanyId,\ Address,\ Name)}(\cup_{i=1}^{i=3} Expand(O_i^*)).$$

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 the 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.
- 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.

## 7. CONCLUSIONS

In this paper, we have presented 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 have proposed a specification for evolving entities and relationships (SPEER). Specifically, SPEER is a high level language for specifying the derivation relationships between schema versions, relationships among attributes, and the conditions for maintaining consistent views of programs. We also developed a three-level schema methodology for mapping changes to the ER diagram into the underlying database and constructing database views for schema versions. Through the reconstruction of views afer database reorganization, changes to an ER diagram can be transparent to the application programs while all objects in the database are accessible to all programs.

In this paper, we restricted our presentation in the context of the basic ER model. However, our approach can be easily generalized to support schema evolution in the Enhanced ER model [19], in which the inheritance relationships (or class hierarchy) may exist among object schemas [12].

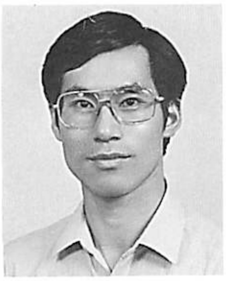
Although SPEER is a powerful high level language as demonstrated by the various examples in this paper, its textual form makes it less attractive and less user friendly. For this reason, we have extended the constructs used in ER diagrams and designed EVolutionary ER (EVER) diagrams, an icon-based language that corresponds to SPEER [13]. We believe that through the manipulation of the graphical icons of EVER diagrams, the specification of changes to an ER diagram can be facilitated and made easy.

As part of our future work, we intend to build a prototype for the exploration of schema evolution in different database models, and also to experiment with schema integration in an attempt to facilitate interoperability in a multi-database system.

## REFERENCES

1. Ahlsen, M. and et. al., "Making Type Changes Transparent," in *Proceedings of IEEE Workshop on Language for Automation*, 1983, pp. 110-117.
2. Banerjee, J., Kim, W., Kim, H., and Korth, H., "Semantics and Implementation of Schema Evolution in Object-Oriented Databases," in *Proceedings of ACM SIGMOD*, May 1987, pp. 311-322.
3. Bertino, E., "A View Mechanism for Object-Oriented Databases," in *Proceedings of 3rd international Conference on Extending Database Technology*, March 1992, pp. 136-151.
4. Bratsberg, S.E., "Unified Class Evolution by Object-Oriented Views," in *Proceedings of the 11th International Conference on Entity-Relationship Approach*, October 1992, pp. 423-439.
5. Chen, P., "The Entity Relationship Model — Toward a Unified View of Data," *ACM Transactions on Database Systems*, Vol. 1, No. 1, 1976, pp. 9-36.
6. Chen, P., "ER vs. OO." in *Proceedings of the 11th International Conference on Entity-Relationship Approach*, October 1992, pp. 1-2.

7. Clamen, S.M., "Schema Evolution and Integration," *Distributed and Parallel Databases: An International Journal*, Vol. 2, No. 1, 1994, pp. 101-126.
8. Clamen, S.M., "Type Evolution and Instance Adaptation," Technical Report, CMU-CS-92-133, School of Computer Science, Carnegie Mellon University, June 1992.
9. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks," *Communications of ACM*, Vol. 13, No. 6, 1970, pp. 377-387.
10. Codd, E.F., "Extending the Relational Database Model to Capture More Meaning," *ACM Transaction on Database Systems*, Vol. 4, No. 4, 1979, pp. 397-434.
11. Kim, W., Bertino, E., and Garza, J.F., "Composite Objects Revisted," in *Proceedings of ACM SIGMOD*, June 1989, pp. 337-347.
12. Liu, C.T., Chrysanthis, P.K., and Chang, S.K., "Database Schema Evolution through the Specification and Maintenance of Changes on Entities and Relationships," in *Proceedings of International Conference on Entity-Relationship Approach*, December 1994.
13. Liu, C.T., Chang, S.K., and Chrysanthis, P.K., "Database Schema Evolution using EVER Diagrams," *in Proceedings of International Workshop on Advanced Visual Interfaces*, June 1994.
14. Liu, C.T., Chang, S.K., and Chrysanthis, P.K., "An Entity-Relationship Approach to Schema Evolution," in *Proceedings of 5th International Conference on Computing and Information*, May 1993, pp. 575-578.
15. Navathe S.B. and Pillalamarri, M.K., "OOER: Toward Making the E-R Approach Object-Oriented," in *Proceedings of the 8th International Conference on Entity-Relationship Approach*, October 1989, pp. 185-206.
16. Segal, M.E. and Frieder, O., "On-the-Fly Program Modification: Systems for Dynamic Updating," *IEEE Software*, Vol. 10, No. 2, 1993, pp. 53-65.
17. Sjoberg, D., "Quantifying Schema Evolution," *Information and Software Technology*, Vol. 35, No. 1, 1993, pp. 35-44.
18. Skarra, H.A. and Zdonik, S.B., "Type Evolution in an Object-Oriented Database," in *Research in Object-Oriented Databases*, Addison-Wesley, 1987, pp. 137-155.
19. Teorey, T., Yang, D., and Fry, J., "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model," *ACM Computing Survey*, Vol. 18, No. 2, 1986, pp. 197-222.
20. Tresch, M. and Scholl, M.H., "Schema Transformation without Database Reorganization," *ACM SIGMOD Record*, Vol. 22, No. 1, 1993, pp. 21-27.
21. Zdonik, S.B., "Object-Oriented Type Evolution," in *Advances in Database Programming Languages*, Addison-Wesley, 1990.
22. Zicari, R.A., "Framework for Schema Updates in an Object-Oriented Database System," in *Proceedings of Conference on Data Engineering*, February 1991, pp. 2-13.



**Chien-Tsai Liu** received his B.S. degree in Electronic Engineering from Tamkang University, Taipei, Taiwan, in 1980, and his M.S. degree in Electronic Engineering from National Taiwan Institute of Technology in 1984. He is currently a candidate for the Ph.D. degree in Computer Science at the University of Pittsburgh. His research interests include design of database management systems, formal software specification, and system prototyping.

**Panos K. Chrysanthis** joined the faculty of the Department of Computer Science at the University of Pittsburgh in 1991, after receiving the M.S. and Ph.D. degrees in Computer and Information Science from the University of Massachusetts. His research interests lie within the areas of database systems, distributed computing, operating systems and real-time systems. In particular, his database research focuses on both formal models and implementation support for structuring reliable distributed systems based on the notions of objects and transactions. The ACTA transaction framework is one of the results of this work. To achieve high performance in database systems, he investigates methods which exploit semantic information to enhance concurrency and reduce the cost of recovery. Currently, he is also investigating issues in mobile and wireless information systems.

**Shi-Kuo Chang** (張系國) received the B.S.E.E. degree from National Taiwan University in 1965. He received the M.S. and Ph.D. degrees from the University of California, Berkeley, in 1967 and 1969, respectively. He is currently Professor and Director of the Center for Parallel, Distributed and Intelligent Systems, University of Pittsburgh. Dr. Chang is a Fellow of IEEE. His research interests include distributed systems, image information systems, visual languages and multimedia communications. Dr. Chang has published over one hundred and seventy papers, and eight books. His books, **Principles of Pictorial Information Systems Design** (Prentice-Hall, 1989), and **Principles of Visual Programming Systems** (Prentice-Hall, 1990), are pioneering advanced textbooks in these research areas. Dr. Chang is the Editor-in-Chief of the *Journal of Visual Languages and Computing* published by Academic Press, and the Editor-in-Chief of the *International Journal of Software Engineering & Knowledge Engineering* published by World Scientific Press.