Enhancing the ER Model with Inegrity Methods

Mira Balaban and Peretz Shoval
Ben-Gurion University of the Negev, Israel

Entity Relationship (ER) schemas include cardinality constraints, that restrict the dependencies among entities within a relationship type. The cardinality constraints have direct impact on application transactions, since insertions or deletions of entities or relationships might affect related entities. Application transactions can be strengthened to preserve the consistency of a database with respect to the cardinality constraints in a schema. Yet, once an ER schema is translated into a logical database schema, the direct correlation between the cardinality constraints and application transaction is lost, since the components of the ER schema might be decomposed among those of the logical database schema.

We suggest to extend the Enhanced ER (EER) data model with integrity methods that take the cardinality constraints into account. The integrity methods can be fully defined by the cardinality constraints, using a small number of primitive update methods, and are automatically created for a given EER diagram. A translation of an EER schema into a logical database schema can create integrity routines by translating the primitive update methods alone. These integrity routines may be implemented as database procedures, if a relational DBMS is utilized, or as class methods, if an object-oriented DBMS is utilized.

INTRODUCTION

Chen (1976) introduced the Entity Relationship (ER) data model as a means for describing in a diagrammatic form, entities and relationships among entities in the subject domain. The ER model enjoys widespread popularity as a tool for conceptual database design, and received many extensions and variations, which are generally termed the Enhanced ER (EER) model. An EER schema can be translated into logical database schemas, usually relational, and implemented with some specific DBMS, using its specific DDL (data definition language). Application programs that manipulate the database access the DBMS via its DML (data manipulation language), either directly or through a host programming language.

EER can be used not only to design a conceptual schema that will later on be translated into a logical schema, but also as a platform for database integration, i.e. to create a meta-schema for a multi data base environment, in which there are heterogeneous databases, utilizing different data models. Co-operation or federation of such databases is possible if a common meta-schema is created. EER can be the high-level model used for that purpose. Similarly, the EER model is used in database re-engineering: the data model of a legacy-system is first reverse-engineered to an EER schema, and later on translated and implemented in a new DBMS. Yet, the EER model deals only with the static (structural) aspects of the data model (namely, entities, relationships and attributes) but not with behavioral aspects (namely procedures to manipulate the data that is defined by the schema, and to preserve the integrity of data). These aspects are taken care of at the implementation level, either by the DBMS (for example, when a relational DBMS performs referential integrity checks), or by the application programs.

An EER schema supports the specification of cardinality constraints, that restrict the dependencies among entities within a relationship type (see, for example, Lenzerini & Santucci, 1983; Lenzerini & Nobili, 1990; Ferg, 1991; Thalheim, 1992]; Thalheim, 1998). For example, the cardinality constraints can specify that a department must have at least five workers and at most eighty. The cardinality constraints have direct impact on maintenance transactions of the target system, since insertions or deletions of entities or relationships might affect related entities. This impact can be captured by operations that a transaction must trigger in order to preserve the cardinality constraints. Yet, once an EER schema is translated into a

Manuscript originally submitted

logical database schema, the direct correlation between the cardinality constraints and maintenance transactions is lost, since the components of the EER schema are usually decomposed among those of the target database schema. Moreover, at this level it is up to application programmers to correctly capture the constraints.

In this paper we suggest to enhance the EER model with the behavioral aspects of the cardinality constraints (see also Lazarevic & Misic, 1991). Specifically, we suggest extending the EER data model with **integrity methods**, i.e. methods that maintain the consistency of data according to the schema definition. Maintenance of consistency means that any attempt to add, delete or change entities and relationships in the subject domain of a schema is checked against the schema definitions, as expressed by cardinality constraints. The integrity methods can be fully defined by the cardinality constraint. Hence, once the enhanced EER schema is mapped to a logical schema, the integrity methods can be mapped to respective integrity routines. These integrity routines may be implemented as database procedures, if a relational DBMS is utilized, or as class methods, if an object-oriented DBMS is utilized. The integrity methods are built on top of **primitive update methods** that perform the update transactions. This separation adds a layer of abstraction that enables to define the mapping of a behavior-enhanced EER schema into some target database schema, in terms of the primitive methods alone.

Another popular data model, which is closely related to the EER model, is the Object-Oriented (OO) model. The OO model supports static data abstraction using *object classes*, and models system behavior through "methods" (procedures) that are attached to object classes, with message passing as a means for communication among objects. Moreover, unlike EER, which is mainly used for conceptual data modeling, the OO approach is also utilized on the OO-DBMS level. Consequently, there is a direct transition from an OO data model to its implementation as an OO-DBMS. Therefore, there is a tendency to assume that the OO approach can/will replace EER as a data modeling tool (see, for example, Kornatzky & Shoval, 1994; Elmasri & Navathe, 1994; Dittrich, 1987).

Nevertheless, the EER model might still be superior for the task of conceptual data modeling, since it enables finer distinctions between entity types to relationship types, and provides an explicit account for the associated constraints. Indeed, the direct transition between the levels of analysis, design and implementation is just one criterion. Other preference criteria among modeling tools involve various perspectives, e.g. learnability, comprehension (i.e., how easy it is for users to understand the schema), and quality of design (i.e., how accurate and complete is the schema that is being designed). Indeed, experimental comparisons show that EER is superior to OO with respect to the criteria of comprehension, quality of design, time needed for completion of a design task, and designers preference of models (Shoval & Frumermann, 1994; Bock & Rian, 1993; Shoval & Shiran, 1997). Shoval & Shiran (1997) concluded that even if the objective is to design and implement an OO schema, within an OO-DBMS or any other OO programming environment, the following strategy is recommended: first, design an EER schema; then map it to an equivalent OO schema. Finally, augment the OO schema with the necessary behavioral constructs (e.g., methods and messages).

A recent effort to standardize all aspects of object modeling is accomplished within the Unified Modeling Language (UML) initiative (Fowler, 1997]; Booch et al, 1999). UML is a collection of visual languages that covers the static data, process and behavioral perspectives in modeling. The data modeling aspect in UML is handled by class diagrams and object diagrams, and also provides an Object Constraint Language. The static part of UML is richer than the conventional object schemas in object-oriented databases and in programming languages. The EER model is an integral part of UML — actually, UML includes all constructs of EER schemas, except for weak entity types. Indeed, we use the EER model as a representative for the static part of object modeling, in general. Our work can be viewed in the wider aspect of extending UML with structure sensitive methods. The extension of the EER data model with methods blurs the difference between OO schemas to EER schemas.

In the following section the EER data model is introduced, and subsequent sections describe the suggested enhancement with structure methods.

THE ENHANCED-ENTITY-RELATIONSHIP (EER) DATA MODEL

EER is a data model for describing entities, their properties, and inter-relationships. A set of entities that share a common structure is captured as an *entity type*. Regular properties of entities are captured as their *attributes*. The attributes are associated with the entity types, and can be either *simple* or *composite*. In most entity types, entities are identified by an attribute (or attributes), called a key (or keys). In some entity types, entities are identified by their inter-relationships to other entities. Such entity types are termed *weak*.

Interactions among entities are modeled by relationships. A *relationship type* relates several entity types; it denotes a set of relationships among their entities. The number of entity types related by a relationship type is its *arity* (≥ 2). The role of an entity type within a relationship type is specified by its *role-name*. Role-names are mandatory in case that an entity type plays several roles within a single relationship type. Two-ary relationship types are called *binary*. A binary relationship type that relates an entity type to itself is called *unary*. Specialization and generalization inter-relationships among entity types are singled out as special kinds of relationships.

The *cardinality constraints* are set on relationship types, and characterize numerical dependencies among entities within

the relationship types. Existing EER models support a variety of cardinality constraints, and sometimes use the same notation with different semantics (Lenzerini & Santucci, 1983; Lenzerini & Nobili, 1990; Ferg, 1991; Thalheim, 1992; Thalheim, 1998). There are two main kinds of cardinality constraints: *participation* and *look-across*. Both kinds of cardinality constraints can be imposed either on the entity types related by the relationship type, or on the entities that already occur in the relationship type. For example: Let's assume a ternary relationship **sale** among the entity types **Salesperson**, **City** and **Product**.

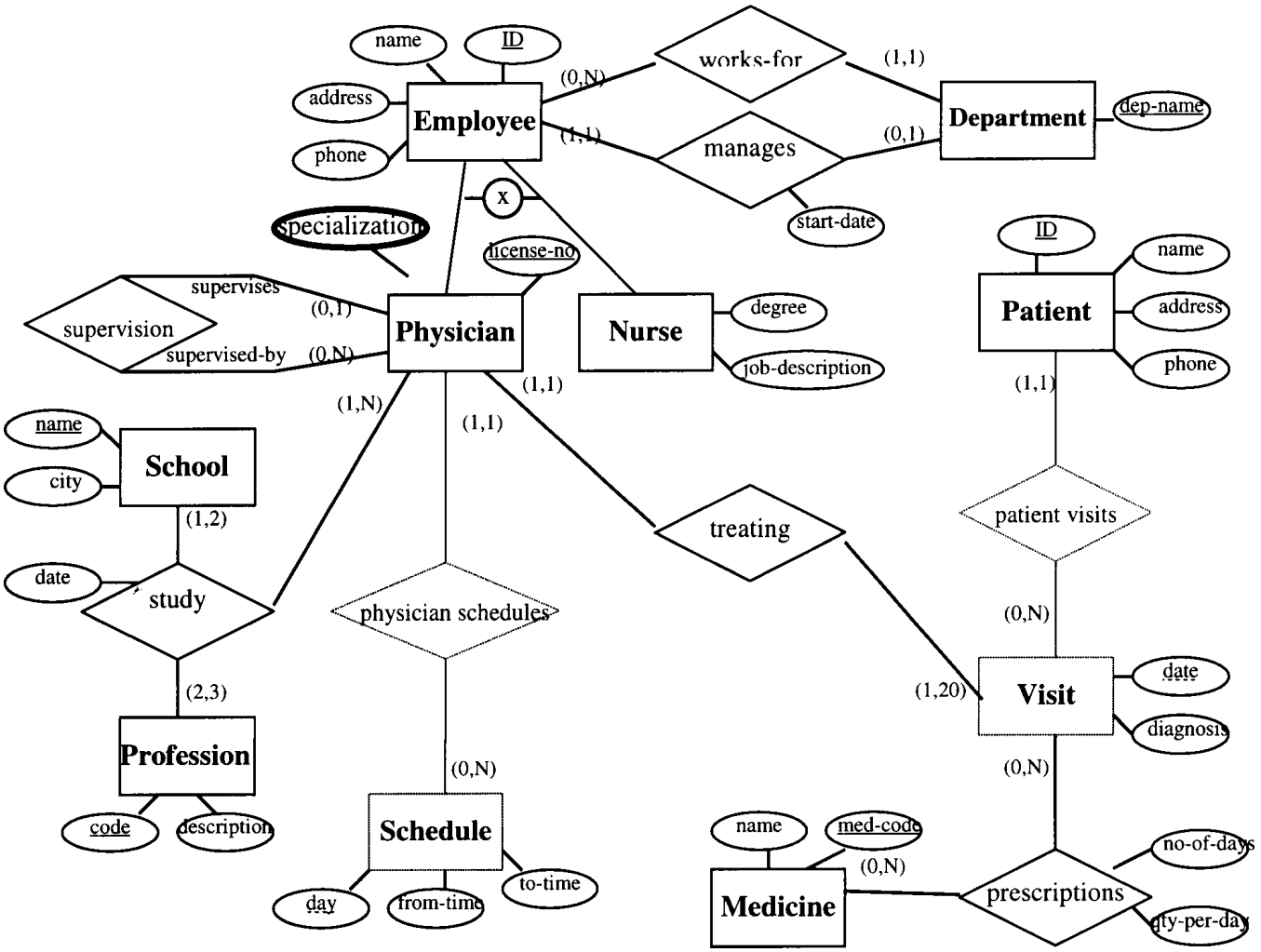
1. Every **Salesperson** entity must be involved in at least three sales events — *a participation constraint* on the entity type **Salesperson**.
2. Every **Salesperson** entity must sell at least two and at most five products in every city — a *look-across constraint* on the entity type **Salesperson**.
3. If a **Salesperson** entity is already active, i.e., involved in some sales event, then it is involved in at least two more sales, and in at most seven — a *participation constraint* on the relationship type sale.
4. Every **Salesperson** entity that sells a product in some city can be constrained to sell that product in at least two more cities — a *look-across constraint* on the relationship type **sale**.

In this paper we consider only look-across cardinality constraints. For binary relationship types, the constraints are imposed on the related entity types (and thus, happen to coincide with participation constraints); for non-binary relationship types the cardinality constraints are imposed on the relationship types themselves.

Figure 1 presents an EER diagram for a medical clinic. Rectangles describe entity types, diamonds describe relationship types, circles describe attributes, solid lines among rectangles describe entity type hierarchies, and dotted line rectangles and diamonds stand for weak entity types and their relationships to the respective owner entity types.

We now turn to a more formal description of the EER

Figure 1: An EER diagram for a medical clinic information system



model (see also Calvanese et al, 1998). An EER schema consists of *entity type* symbols, *relationship type* symbols (each with associated arity), *role name* symbols, and *attribute* symbols. Entity type symbols can be strong (denoting non-weak entity types) or *weak*. Attribute symbols can be *simple* or *composite*. A composite attribute symbol is associated with a set of attribute symbols (other than itself). Furthermore, each attribute symbol is either *single-valued* or *multi-valued*.

Every entity type symbol is associated with a set of attribute symbols, called its *attributes*. An attribute symbol can be singled out as a *key*. A strong entity type symbol has at least one *mandatory key*. A relationship type symbol can also have attributes, and may be associated with an optional key. Every relationship type symbol of arity n is associated with n entity type symbols, each with an associated role name symbol, and minimum and maximum cardinality constraints. This complex association is captured by the syntactic relationship construct

$$R(RN_1 : E_1[min_1, max_1], ..., RN_n : E_n[min_n, max_n])$$

where E_i ($1 \leq i \leq n$) are entity type symbols, RN_i ($1 \leq i \leq n$) are role names, and min_i and max_i are natural numbers or the special symbol ∞. For example, the above look-across constraint on the relationship type **sale**: "Every **Salesperson** entity that already sells a product in some city is constrained to sell that product in at least two more cities and at most five" is captured by the relationship construct:

sale(Salesperson : **Salesperson**[1, ∞],
product : **Product**[1, ∞], city : **City**[3, 5])

The role names RN_i are used to identify the components of the relationship constructs. That is, $RN_i(R) = E_i$, for $1 \leq i \leq n$. In practice, role names are optional; if they are not provided within the schema, then they are schema created. A conventional simplified cardinality notation uses 1 for $min_i = max_i = 1$, and a letter (e.g., n) for $min_i \geq 0$, $max_i = \infty$. So we get cardinality constraints such as 1:n:m, 1:n, 1:1, etc.

An entity type symbol may be associated with a set of entity type symbols (other than itself) that form its *specialization* or *sub-typing*. The entity type symbol is called the *super-type* of the specialization, and the associated entity type symbols are called its *subtypes*. The super-type and subtype relations are extended to include their transitive closure. The sets of attributes of an entity type and of any of its super-types are disjoint (no over-writing). The specialization has a *kind*, which is one of three values X, T, XT. The kind X marks disjointness of the specialization subtypes, the kind T marks that the specialization covers the whole super-type, and the kind XT marks both. An entity type symbol may participate, as a subtype, in at most a single specialization.

A key of a type is a means for identifying the instances of the type via their attributes. A key of a strong entity type symbol and of a relationship type (if any) is an attribute of the type. A key of a weak entity type symbol is defined through related *owner* entity type symbols. Every weak entity type symbol E is associated with one or more binary relationship type symbols R_1 ,... , R_k, termed the identifying relationship types of E, such that for each identifying relationship type R_i of E, the EER schema includes a relationship construct:

$$R_i(RN : E[n, m] , RN' : owner_i(E)[1, 1]).$$

Each entity type symbol $owner_i(E)$ is termed an *owner entity type* of E for R_i. The cardinality constraints for E in the relationship constructs for its identifying relationships mean that every entity of E is associated with a single owner entity, for every identifying relationship. The key of E consists of the keys of its owners, and its own partial key (if exists), which is any of its attributes. That is, $key(E) = (key(owner_1(E)), ... , key(owner_k(E)), A)$, where A is an attribute of E.

A database instance D of an EER schema ER is defined by a non-empty finite domain of entities D, a domain assignment *dom* for the attributes, and a meaning assignment for the symbols of the schema. *dom* is a partial mapping that associates a pair (A,T) of a simple attribute symbol A and a type symbol T (entity or relationship) with a value domain $dom(A,T)$. dom is extended to composite attributes by defining $dom(A,T) = dom(A_1,T) \times ... \times dom(A_n,T)$ for a composite attribute symbol A that is associated with the attribute symbols A_1, ..., A_n. The legal values of an attribute A of a type T are the values in $dom(A,T)$. The application of the meaning assignment to a symbol s of ER is denoted s^D. It is defined as follows:

1. For an entity type symbol E, E^D is an entity type, i.e., a subset of D. The elements of E^D are entities.
2. For a relationship type symbol R with arity n, R^D is a relationship type, i.e. an n-ary relation over D. The elements of R^D are relationships, and their components are labeled with role names. That is, instead of viewing relationships as ordered tupels, they are rather viewed as sets of labeled components. If the role names in the relationship construct of R are RN_1, ... , RN_n, then we refer to the relationships in R^D as sets of the form $r = \{RN_1 : e_1, ..., RN_n : e_n\}$, We define $RN_i(r) = e_i$, ($1 \leq i \leq n$). The role name symbols RN_1, ... , RN_n are refered to as the roles of R^D.
3. For an attribute symbol A of a type symbol T (entity or relationship), $(A ,T)^D$ is an attribute of T^D, i.e., a partial function from T^D into either dom(A,T) — if A is single-valued, or into the power set of dom(A,T) — if A is multi-valued.

A database instance of a schema ER is consistent if it satisfies the intended meaning of keys, relationship constructs, cardinality constraints, and sub-typing relationships. An EER schema is *consistent* if it has a consistent database instance. The constraints set by keys mean, for strong entity types and for relationship types, that the key attribute values uniquely identify the instances of the types. For weak entity

types, their owner entities and their key attribute values uniquely identify them. A sub-typing of E by $E_1, \ldots, E_n$ in ER, constrains E_i^D to be a subset of E^D $(1 \le i \le n)$. If the kind of the specialization is X, then the sets E_i^D are disjoint; if the kind is T, then the sets E_i^D cover the set E^D; if the kind is XT, then both constraints hold. Note that these constraints imply inheritance of attributes and relationships through specialization relationships.

A relationship construct $R(RN_1 : E_1[min_1, max_1], ..., RN_n : E_n[min_n, max_n])$ in ER imposes the following constraints:

1. $R^D \subseteq E_1^D \times \ldots \times E_n^D$.
2. The cardinality bounds on the i-th component delimit the number of elements in EiD that can be related to given elements of the other entity types. The meaning for binary and non-binary relationship types is different: In binary relationship types the cardinality constraints in one entity type apply to every entity of the other entity type, while in non-binary relationship types the cardinality constraints in the i-th component apply only to every R^D-related combination of entities from the other entity types.

   One. **Binary relationship types**:

   For i = 1,2 and j = 2,1, respectively:

   For all e_j in E_j^D ,

   $min_i \le$ **cardinality**$(\sigma_{RNj = ej} (R^D)) \le max_i$.

   For example, in Figure 1 the constraint

   supervision(*supervises* : **Physician**[0, 1], *supervised-by* : **Physician**[0, ∞])

   means that in every consistent database instance D, for every **Physician**^{D} entity p, selection of p on the role name *supervised-by* in the relationship type **supervision**^{D}, yields at most a single relationship.

   Two. **Non-binary relationship types**:

   For all $\{ RN_1 : e_1, \ldots, RN_n : e_n \}$ in R^D:

   For all $1 \le i \le n$,

   $min_i \le$ **cardinality**$(\sigma_{\{RN1 = e1, \ldots, RNi-1 = ei-1, RNi+1 = ei+1, \ldots RNn = en\}} (R^D) \le (max\_i$.

   For example, the constraint

   sale(*Salesperson* : **Salesperson**[1, ∞], *product* : **Product**[1, ∞], *city* : **City**[3, 5])

   means that in every consistent database instance D, for every **sale**^{D} relationship with a Salesperson s and a product p, selection of s and p on the role names *Salesperson* and *product*, respectively, in the relationship type **sale**^{D}, yields between three to five relationships.

EER EXTENDED WITH METHODS

We suggest extending the EER schema with integrity methods, which are update methods that are sensitive to the cardinality constraints. The integrity methods should be defined on top of primitive update methods, which are integrity insensitive. The rational behind this separation is that of the Abstract Data Types (ADT) approach: The primitive update methods serve as an *abstraction barrier* between the integrity methods and a logical database implementation of the EER schema. Every such implementation can be defined in terms of the primitive update methods alone. The integrity methods stay intact. The advantage of this approach is that the integrity methods are defined, and their properties are proved, once and for all on the EER level. Since the definitions and proofs are rather complex, the advantage is clear.

Primitive Methods

The primitive methods perform *insertion, deletion, retrieval* and *attribute modification* in a database instance D of an EER schema ER. They are associated with the entity and the relationship type symbols of the schema. Insertions always involve the creation of a new entity or relationship. Consequently, in a database instance created only with the primitive methods, entity types that are not related by the subtype/supertype relation have no entities in common. Similarly, all relationship types are mutually disjoint.

The primitive methods should be sensitive to the subtype and super-type relations in the sense that an insertion of an entity to the entity type E^D inserts it also to all super entity types of E^D . Similarly, deletion of an entity from E^D deletes it also from all sub entity types of E^D.

The addition of methods requires operations for retrieving the components of an EER schema, and of the information associated with instances in a database instance of a schema. These operations can be denoted as follows:

1. **Schema level operations**: For an entity type symbol E, the relationship type symbols that their constructs involve E or a super type symbol of E, and the corresponding role names, are given by E.*rels* = { [R , RN] | RN(R) = E', E' = E or is a super type symbol of E }. For a relationship type symbol R, the role names are given by R.*role_names*.
2. **Database level operations**:

   One. For an entity e of E^D, e.A retrieves the value on e of attribute $(A, E')^D$, where E' is either E or a super-type of E). e.A is uniquely defined since the sets of attribute symbols associated with E and its super-types are mutually disjoint. A *legal key value* for A of E in D is a value in the domain that D assigns to a key attribute symbol A of E. For every [R , RN] in E.rels, e.relationships([R , RN]) are the R^D relationships whose RN component is e, and *e.no_of_relationships*([R , RN]) is their number.

   Two. For a relationship r of R^D, r.A retrieves the value of attribute $(A, R)^D$ on r. r.RN retrieves the RN entity component of r, for every role name RN in R.*role_names*. A *legal relationship* for R in D is a labeled set $\{ RN_1 : e_1, \ldots, RN_n : e_n \}$, such that R.*role_names* = $\{ RN_1, \ldots, RN_n \}$, and ei is an entity in an entity type identified by the role name RN_i.

Primitive methods for an entity type symbol E: v is a legal key value for A of E in D.

1. E.**insert** (A : v) — Creates a new E^D entity e, such that e.A

= v, and for every [R, RN] in E.*rels*, *e.no_of_relationships*([R, RN]) = 0, and *e.relationships*([R, RN]) = ϕ. The entity e is added to all super entity types of E^D. The return value is e.

2. E.**delete** (A : v) — Deletes from E^D and from all entity types E'^D such that E' in E.*subs*, the entity e identified by the value v of $(A, E)^D$ (i.e., e.A = v), if any.
3. E.**retrieve** (A : v) — Retrieves from E^D the entity e identified by e.A = v, if any. The return value is either e, or NULL, if there is no such entity.
4. E.**retrieve_all**() — Retrieves all entities in E^D.

Primitive methods for a relationship type symbol R: Let r = { $RN_1 : e_1$, ... , $RN_n : e_n$ } be a legal relationship for R in D.

1. R.**insert**(r) — Creates an RD relationship r with $r.RN_i = e_i$ $(1 \le i \le n)$. The return value is r. The entities ei $(1 \le i \le n)$ are updated as follows: e_i.no_of_relationships([R, RN_i]) is increased by one, and r is added to e_i.relationships([R, RN_i]).
2. R.**delete**}(r) — Deletes from R^D the specified relationship, if any. If there is a deletion, the entities ei $(1 \le i \le n)$ are updated to decrease e_i.no_of_relationships([R, RN_i]) by one, and remove r from e_i.relationships([R, RN_i]).
3. R.**retrieve** (r) — Retrieves r from R^D, if any. The return value is either r, or NULL, if there is no such relationship.
4. R.**retrieve_all**}() — Retrieves all relationships in R^D.

Primitive methods for attribute modifications:

These methods perform modification, removal, and retrieval of a value of an attribute of an instance of a type. Simultaneous modification of multiple attributes can be handled by introducing compound update methods. Let T be a type symbol (entity or relationship). If T is an entity type symbol, let v be a legal key value for T in D. If T is a relationship type symbol, let v be a legal relationship for T in D. Let A be an attribute symbol associated with T, and val a legal attribute value for (A, T) in D, i.e., *val* belongs to *dom*(A, T).

1. T.**modify**(v, A, val) — If A is a single-valued attribute symbol, val is substituted for any previous value of the attribute $(A, T)^D$ on the instance (entity or relationship) identified by v. If A is a multi-valued attribute symbol, val is added to any previous value of the attribute $(A, T)^D$ on the instance identified by v.
2. T.**remove**(v, A, val) — If A is a single-valued attribute symbol, and the value of the attribute $(A, ,T)^D$ on the instance identified by v, is val, it is replaced by NULL. If A is a multi-valued attribute symbol, val is removed from the value of the attribute $(A, T)^D$ of the instance identified by v.
3. T.**get**(v, A) — Retrieves the value of attribute $(A, T)^D$ of the instance identified by v.

Integrity-Preserving Policies

In order to preserve the consistency of the database, an integrity update method might invoke associated update methods or be refused. We distinguish four integrity-preserving policies:

1. **Reject** — the update operation is refused. This is in some sense a brute force action for integrity preservation. It should only be used with caution, in order to not block database updates.
2. **Propagate** — an insertion or deletion of an instance violates a cardinality constraint, and invokes appropriate deletion or insertion actions. Propagation is achieved by dispatching the impact of a newly inserted or deleted entity or relationship to its neighboring relationships. Among the four integrity-preserving actions, **Propagate** is the most faithful to the policy of integrity preservation. But it is also the most expensive, and one should be careful not to embark on an unlimited sequence of update operations. Since the schema is consistent, it has consistent (finite) database instances. In general, it is worthwhile that the user can guarantee full propagation before actual updates are applied.
3. **Nullify** — violation of a cardinality constraint is relaxed by the insertion of a new **null** *entity*, and including it in a relationship. **Nullify** is a compromise between the desire to preserve integrity, and the inability or unwillingness to propagate. In a **Nullify** operation a "fictional" entity is inserted to an entity type and connected to "real" entities, so that their integrity is preserved. The assumption is that cardinality constraints do not apply to null entities. A null entity can be replaced by a real one by reconnecting its related entities to a real entity.
4. **Schema revision** — integrity violation is removed by revising, or re-defining, the cardinality constraints. The revision can only decrease a minimum cardinality constraint, or increase a maximum cardinality constraint. **Schema revision** is intended to resolve impossible cardinality constraints, or emerges from new definition of the domain. It seems that one should be careful not to abuse this intention by using this action as a replacement for simple **Propagate** or for **Nullify** so to temporarily preserve all constraints.

These integrity-preserving policies represent conventional approaches for integrity maintenance (Etzion & Dahav, 1998). We suggest that these policies should be determined in an interactive mode, and not be fixed in advance for the different types of the schema (as suggested in Lazarevic & Mimic, 1991).

Integrity Methods of Entity Types

The **integrity_insert** and **integrity_delete** operations might invoke the **Propagate** policy for integrity preservation. Propagation for insertion is caused by non-zero minimum constraints in binary relationship constructs, since they imply

that a newly added entity must be related to another entity. Minimum constraints in non-binary relationship constructs do not pose any restriction on a new entity, since they apply only to already existing relationships. Propagation for deletion is caused by relationships in which the deleted entity participates. Maximum constraints are not violated by insertions or deletions of entities, but should be considered when an already existing entity is connected to a new relationship (see next subsection).

I. The Integrity_Insert Method

The **integrity_insert** method for an entity type symbol E, and a legal key value v for A of E in a database instance D, involves the following operations: If A : v does not identify an already existing entity in E^D, a new entity with the key value v for A is inserted into E^D, and the insertion effect is propagated. The propagation involves observation of all binary relationship symbols with which E or any ancestor of E is associated. If the minimum cardinality constraints that they set on an entity in E^D are not met by the new entity, then the involved relationship types are asked to provide the missing relationships with the new entity. This way the insertion is propagated from E to its related relationship type symbols, and from there it can further propagate to new entity type symbols.

Example:

Consider the EER schema from Figure 1. In order to insert a **Physician** entity with a license number ph12345 to a database instance, **Physician.integrity_insert**(license-no : ph12345) is applied. Integrity preservation requires that several constraints be checked, before the real insertion takes place. First, we need to check that this physician is not already in the database. Second, we need to observe all binary relationship type symbols whose constructs involve **Physician** or any ancestor of **Physician**, to see whether their cardinality constraints are not violated. The binary relationship type symbols **supervision, physician-schedules** and **manages** do not constrain the new entity since they have zero minimum constraints.

The binary relationship type symbols **treating** and **works-for** provide minimum requirements on the number of relationships involving each physician instance: Every physician must have at least one visit, and must work for some department. So, we have to ask the user for a candidate visit to be related to the new physician through the treating relationship type, and for a candidate department entity as well. The user might provide an already existing entity, or suggest a new one. Indeed, the required department entity may be an already existing one, but the required visit must be a new entity, since for any visit there is exactly one physician. So, every visit entity already in the database, already has its physician related to it through **treating**. Once the user provides a new **Visit** entity, the process of integrity preservation has to repeat itself. The new visit might not be related to a **Medicine** entity through **prescriptions**, but it must have a **Patient** entity, through **patient-visits**. Again we need to ask the user for a candidate patient, which in this case can be an old or a new patient. In any case, since there are no minimal restrictions on **patient-visits** relationships, the propagation stops. In order to avoid or terminate propagation, the user might decide to **Nullify** missing entities in relationships. For example, if the department of the new physician does not exist in the database yet, the user can insert a new null entity to the **Department** entity type, and connect it to the new physician in a **works-for** relationship. Later on, when the missing department is inserted, the user can reconnect the current physician entity to it.

II. The Integrity_Delete Method

The **integrity_delete** method for an entity type symbol E, and a legal key value v for A of E in a database instance D, involves the following operations: If A : v indeed identifies an already existing entity in E^D, then the entity is deleted, and the deletion effect is propagated. The propagation involves observation of all relationships that include the removed entity. These relationships can be instances of relationship types R^D, for R in E.rels. For each such relationship, the user can choose one of four possibilities: **Reject** the update, **Nullify** the references to the removed entity, **Reconnect** the relationship, or **Delete** the relationship. In the **Reconnect** option, the user is asked to replace the deleted entity with another one, new or old. Insertion of a new entity might propagate as above. In the **Delete** option, the relationship is deleted, and the effect may propagate, either to further deletion of the related entity (for a binary relationship), or to further deletions of other relationships in the same relationship type (for a non-binary relationship). Note that it may not be possible to avoid propagation by deleting the relationships before the entities, since the deletion of a binary relationship is sanctioned, if it violates the minimum constraints set on the related entities.

Example:

Consider the EER schema from Figure 1. In order to delete the **Physician** entity with license number ph12345 from the database, **Physician.integrity_delete**(*license-no* : ph12345) is applied. Integrity preservation requires that several constraints be checked, before the real deletion takes place. First, we need to check that this physician is indeed in the database. Second, we need to decide what to do with each relationship in which this entity participates. As explained above, we can choose among **Reject, Nullify, Reconnect** or **Delete**. For example, due to the cardinality constraints on the treating relationship type, the physician entity must have at least one treating relationship. If we decide to reconnect the relationship, it might be to an already existing physician, or to a new physician. For an existing physician, we need to check whether it does not exceed the maximum of 20 visits. For a new physician, it should be inserted, connected, and the effect of the insertion propagated.

If we decide to delete the **treating** relationship, the minimum cardinality constraints on the **Visit** entity type

should be checked. In general, if a **Visit** entity could be related to several **Physician** entities, then it might have been possible to delete the relationship without violating the cardinality constraints. However, here, since every visit must be related to exactly one physician, deletion of the relationship violates the database integrity, and must invoke propagation of the deletion to the related visit. In order to avoid or terminate the propagation, the user might decide to **Nullify** the **Physician** entity in the **treating** relationship of that visit.

Integrity Methods of Relationship Types

The operations are **connect, disconnect**, and **reconnect**. The **connect** operation inserts a new relationship between existing entities, the **disconnect** operation deletes a relationship, and the **reconnect** operation replaces an entity in a relationship. Our policy is to restrict the **connect** and **disconnect** operations so that they do not propagate outside the relationship type under consideration. If they violate the integrity of the related entities (for binary relationship types only) they are rejected, and can be replaced by other operations, such as **reconnect**, or **integrity_insert** or **integrity_delete**. The **reconnect** operation, on the other hand, can propagate to the related entity types, when an entity in a relationship is replaced by a new entity. This propagation cannot be avoided, since the database integrity might block the independent insertion of the new entity.

I. The Connect Method

The **connect** method for a relationship type symbol R, and a legal relationship $r = \{ RN_1 : e_1 , ... , RN_n : e_n \}$ for R in a database instance D, involves the following operations: If r does not already exist in R^D, and if all entities ei $(1 \le i \le n)$ already exist, the new relationship is tested not to violate the cardinality constraints set on R. If the test succeeds, the new relationship is inserted to R^D.

Example:

Consider the EER schema from Figure 1. In order to connect a **Physician** entity p to a **Visit** entity v, **treating.connect**({**treating.physician** : p, **treating.visit** : v}) is applied (**treating.physician** and **treating.visit** are the schema provided role names for the **treating** relationship type). If both entities do exist in the database instance, and can be connected without violating the maximum cardinality constraints set on **treating**, the relationship is inserted to the **treating** relationship type. Note that the maximum constraint on the **visit** entity v is 1. That means that for the connection to be performed, v must have been inconsistent prior to the connection. This can be the case, for example, if the current **treating.connect** operation was invoked from within **Visit.integrity_insert**(**Patient**.*ID* : p123, date : 3.1.98), assuming that v is identified by the key value (**Patient**.*ID* : p123, date : 3.1.98). No further updates are invoked since **treating** is binary.

In order to connect a **Physician** entity p to a **Profession** entity f, and a **School** entity s, **study.connect**({**study.physician** : p, **study.profession** : f, **study.school** : s}) is applied. If the entities do exist in the database instance, and can be connected without violating the maximum cardinality constraints set on **study**, the relationship is inserted to the **study** relationship type. Since **study** is non-binary, an auxiliary method, **study.make_consistent**({**study.physician** : p, **study.profession** : f, **study.school** : s}) should be applied. Assume, for example, that there are no other **study** relationships with the physician p and the school s. Then, one such **study** relationship is still missing (there should be at least 2, and at most 3). Then the user can be asked to choose among **Reject, Nullify, Connect** or **Schema revision**. If **Nullify** is chosen, then a new null entity is inserted to the **Profession** entity type, and connected to the entities p and s to yield a new **study** relationship. If **Connect** is chosen, an existing or a new **Profession** entity is connected to p and s to yield a new **study** relationship. If the **Profession** entity is new, it should also be made consistent (by invoking **Profession.make_consistent** on it).

II. The Disconnect Method

The disconnect method for a relationship type symbol R, and a legal relationship $r = \{ RN_1 : e_1, ... , RN_n : e_n \}$ for R in a database instance D, involves the following operations: If r exists in R^D, and if the deletion of the relationship does not violate minimum cardinality constraints set on the participating entity types in R (possible only if R is binary), then the relationship is disconnected, and the effect is propagated to other relationships in R^D, if needed. The method should treat differently binary relationship types and non-binary ones. For a binary R, preserving the integrity of its cardinality constraints might require deletion of the related entities (since the constraints are imposed on the related entities). We feel that in such cases, it is more reasonable to start with the deletion of the related entities and not with their relationship. Hence, violation of the cardinality constraints for a binary R leads to rejection. For a non-binary R, violation of its cardinality constraints might require reconnection or disconnection of other relationships in R^D. is applied to every selection of n-1 entities from r, to see whether they still meet the minimum cardinality specified for the n-th entity type symbol.

Example:

Consider the EER schema from Figure 1. In order to disconnect the **treating** relationship {**treating.physician** : p, **treating.visit** : v}, the method **treating.disconnect**({**treating.physician** : p, **treating.visit** : v}) is applied. If the **Visit** entity is maximally consistent (i.e., v is related only to the **Physician** entity p), the **disconnect** should be rejected, since it violates the minimum cardinality constraints set on **Visit**. In any case, since **treating** is binary, no further updates are invoked.

In order to disconnect the **study** relationship {**study.physician** : p, **study.profession** : f, **study.school** : s},

the method **study.disconnect**({**study.physician** : p, **study.profession** : f, **study.school** : s}) is applied. First, the relationship should be deleted from the **study** relationship type. Since **study** is non-binary, an auxiliary method

study.make_deleted_relationship_consistent
(**study.physician** : p, **study.profession** : f, **study.school** : s)

should be applied. Assume, for example, that prior to the disconnection, there were exactly two study relationships with the physician p and the school s. Then, following the disconnection, one such relationship is missing. The user can be asked to choose among **Reject, Nullify, Connect, Disconnect**, or **Schema revision**. If **Nullify** is chosen, then a new null entity is inserted to the Profession entity type, and connected to the entities p and s instead of the **study** relationship that was just deleted. If **Connect** is chosen, an existing or a new **Profession** entity is connected to p and s to yield a new **study** relationship. If the **Profession** entity is new, it should also be made consistent (by invoking **Profession.make_consistent** on it). If **Disconnect** is chosen, then the remaining **study** relationship with physician p and school s, is disconnected. If **Schema revision** is chosen, then the minimum bound on the **study.profession** role of the **study** relationship type is decreased (from 2 to 1).

III. The Reconnect Method

This operation stands for a **disconnect** operation that is followed by a **connect** operation. However, under the integrity preservation policy, **reconnect** is essential since otherwise, there would be no way to reconnect a relationship that includes an entity with mandatory participation, and participates in no other relationship. For example, if Fred is an employee that moves from the Eye-Department to the Internal-Department, disconnecting the (Fred, Eye-Department) relationship is rejected due to the mandatory participation of **Employee** in the **works-for** relationship type. Hence, the move cannot be achieved as a sequence of **disconnect** and **connect**.

The **reconnect** method for a relationship type symbol R, accepts three parameters: A legal relationship for R in a database instance D, a role name RN for R, and a legal key value A : v for the super entity type symbol identified by RN in R given by R.E_of_RN, with respect to D. For simplicity, we denote the relationship r = { RN : e, RN_1 : e_1 , ... , RN_k : e_k } , where RN is the role name whose entity e should be replaced by the new entity that is identified by A : v. The **reconnect** method involves the following operations: If r exists in R^D, and if the deletion of the relationship does not violate minimum cardinality constraints on the replaced entity e (possible only if R is binary), then the relationship is disconnected, the effect is propagated to other relationships in R^D, and if the insertion of the new relationship does not violate maximum cardinality constraints set on R, the relationship is reconnected. This last test refers to the super entity type symbol identified by the role name RN in R (given by R.E_of_RN). First, it is checked whether A : v identifies an already existing entity in $(R.E\_of\_RN)^D$. In that case, if this entity can be connected to the entities e_1 , ... , e_k to form a new relationship r' in R^D, r' is inserted to R^D. If not, the user is asked to replace A : v by another legal key value. If A : v does not identify an entity in $(R.E\_of\_RN)^D$, a new entity with the key value A : v is inserted to $(R.E\_of\_RN)^D$, the new relationship is inserted to R^D, and the effect of the entity insertion is propagated in D.

Example:

Consider the EER schema from Figure 1. In order to reconnect the **treating** relationship {**treating.physician** : p, **treating.visit** : v}, to a new physician, identified by the key value *license_no : ph5678* of **Physician**, the method **treating.reconnect**({**treating.physician** : p, **treating.visit** : v}, **treating.physician**, *license_no : ph5678*) is applied. If v is the only visit of the physician entity p, then after the deletion of {**treating.physician** : p, **treating.visit** : v }, p becomes inconsistent as it must have at least one visit. The user can be asked to compensate for the missing **treating** relationship of p either by connecting it to a **Visit** entity (a new one — real or null, or an already existing one), or by deleting p. After that, the physician p', identified by the license number ph5678 can be inserted to **Physician** (if it is not already there), the new **treating** relationship { **treating.physician** : p' , **treating.visit** : v } can be connected, and p' should also be made consistent if it is new.

In order to reconnect the **study** relationship {**study.physician** : p, **study.profession** : f, **study.school** : s}, to a new physician, identified by the key value *license_no : ph5678* of **Physician, study.reconnect**({**study.physician** : p, **study.profession** : f, **study.school** : s}, **study.physician**, *license_no : ph5678*) is applied. Since **study** is non-binary, the deletion of the relationship {**study.physician** : p, **study.profession** : f, **study.school** : s} does not affect any entity inconsistency. Then the physician p', identified by the license number *ph5678* is inserted to **Physician** (if it is not already there), the new **study** relationship {**study.physician** : p' , **study.profession** : f, **study.school** : s} can be connected, and p' should also be made consistent if it is new. Since **study** is non-binary, the effect of the deleted **study** relationship should also propagate to the rest of the **study** relationships.

Consistency preserving property of the integrity methods:

The integrity methods suggested in this paper are valuable since they can preserve the integrity of a consistent database instance. That is, if D is a consistent data base instance of an EER schema ER, and D' results from D by the application of an integrity method, then D' is a *consistent up to null entities* instance of ER. That is, D' satisfies all key, relationship construct and sub-typing constraints, and all cardinality constraints set on real entities. If D' includes no null entities, then it is a consistent instance of ER.

Integrity Methods of Attributes

The attribute operations are defined for an entity or a relationship type symbol T, and an attribute A of T. They accept the same parameters as their corresponding primitive methods. In principle, they can be extended to accept any number of attribute-value pairs. The attribute operations that we suggest are: T.**Integrity_modify**(v, A, val), T.**Integrity_remove**(v, A, val), and T.**Integrity_get**(v, A,).

CONCLUSION

In this paper we briefly described an extension of the EER data model with integrity sensitive update methods. For that purpose, we first classified the cardinality constraints, and formalized their semantics. The integrity methods can be fully defined by the cardinality constraints, using the primitive update methods. Hence, our approach enables the development of a tool that creates the integrity methods automatically, for a given EER diagram.

We are currently extending the conventional EER-to-OO schema mappings to map the newly added methods. We have proved that the method mapping preserves their semantics. The method mapping is defined in terms of the primitive methods alone. The mapping of the more complicated integrity methods is directly obtained from the primitive methods mapping. Alternative mappings of the EER schema into different logical database schemas can be extended similarly, by faithful translation of the primitive methods. We are also developing a tool that implements this mapping.

The contribution of this research is in capturing the intended meaning of the cardinality constraints as an integral part of the EER schema. Moreover, since the EER schema language is an integral part of the UML language, and since the latter does not account for active integrity maintenance of the associated constraints, the suggested enhancement actually extends the static part of UML with integrity preserving methods. The data-model independence of our approach is, therefore, essential, since a UML conceptual schema should not be hard-wired to a specific logical database model.

REFERENCES

Bock, D. & Ryan, T. (1993). Accuracy in modeling with extended entity relationship and object oriented data models. *J. of Database Management*, 4(4), 30-39.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications* (2nd Edition). Addison- Wesley.

Booch, G., Rumbaugh, J. & Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison-Wesley.

Calvanese, D., Lenzerini, M. & Nardi D. (1998). Description logics for conceptual data modeling. In: Chomicki, J. & Saake, G. (Editors), *Logics for Databases and Information Systems.* Kluwer Academic Publishers.

Chen P. P. (1976). The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1), 9-36.

Dittrich K.R. (1987). Object-oriented database systems. In: Spaccapietra, S. (Editor), *Entity- Relationship Approach*, 51-66. Elsevier Science Publishers.

Elmasri, R. & Navathe, S. (1994). *Fundamentals of Database Systems*. Benjamin/Cummings Publishing Company.

Etzion, O. & Dahav, B. (1998). Patterns of self-stabilazation in database consistency maintenance, *Data and Knowledge Engineering*, 28(3), 299-319.

Ferg, S. (1991). Cardinality concepts in entity-relationship modeling. In: Teorey, E. (Editor), *Entity-Relationship Approach.* North-Holland.

Fowler, M. (1997). *UML Distilled*. Addison-Wesley.

Kornatzky, Y. & Shoval, P. (1994). Conceptual design of object-oriented database schemas using the binary-relationship model, *Data and Knowledge Engineering*, 14, 265-288.

Lazarevic, B. & Misic. V. (1991). Extending the entity-relationship model to capture dynamic behavior. *European Journal of Information Systems*, 1(2), 95-106.

Lenzerini M. & P. Nobili. "On the satisfiability of dependency constraints in entity-relationship schemata". *Information Systems*, 15(4): 453—461, 1990.

Lenzerini, M. & Santucci, G. (1983). Cardinality constraints in the entity-relationship model. In: Davis, Jejodia, Ng, & Yeh (Editors), *Entity-Relationship Approach.* North-Holland.

Shoval, P. & Frumermann. I. (1994). OO and EER conceptual schemas: a comparison of user comprehension. *J. of Database Management*, 5(4), 28-38.

Shoval, P. & Shiran, S. (1997). Entity-relationship and object-oriented data modeling - an experimental comparison of design quality. *Data & Knowledge Engineering*, 21, 297-315.

Thalheim, B. (1992). Fundamentals of cardinality constraints. In: *Entity-Relationship Approach*, 7-23. North-Holland.

Thalheim, B. (1998). *Fundamentals of Entity-Relationship Modeling*. Springer-Verlag.

Peretz Shoval is Professor of Information Systems at Ben-Gurion University, Israel, and head of the Information Systems Engineering Program. He received B.A in Economics and M.Sc. in Information Systems from Tel-Aviv University, and Ph.D. in Information Systems from the University of Pittsburgh (1981). Prior to moving to academia he held professional and managerial positions in computer companies and in the IDF. Shoval's research interests include data modeling, information systems analysis and design methods, and information retrieval and filtering.

Mira Balaban is Senior Lecturer of Computer Science and Information Systems at Ben-Gurion University, Israel. She received B.Sc. in Mathematics and Statistics from Tel-Aviv University, and M.Sc. and Ph.D. in Computer Science from Weizman Institute of Science (1982). Balaban's research interests include conceptual modeling, knowledge representation, temporal databases, description logic and conceptual tools for music research.