

# MEER – An EER Model Enhanced with Structure Methods\*

Mira Balaban and Peretz Shoval

Department of Information Systems Engineering

Ben-Gurion University of the Negev

P.O.B. 653, Beer-Sheva 84105

ISRAEL

mira@cs.bgu.ac.il    shoval@bgumail.bgu.ac.il

(972)-7-6472222    (972)-7-6472221

## Abstract

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

In this paper, a full solution to the enforcement of cardinality constraints in EER schemas is given. We extend the Enhanced ER (EER) data model with structure-based update methods that are fully defined by the cardinality constraints. The structure methods are provably *terminating* and *cardinality faithful*, i.e., they do not insert new inconsistencies and can only decrease existing ones. A refined approach towards measuring the cardinality consistency of a database is introduced. The contribution of this paper is in the automatic creation of update methods, and in building the formal basis for proving their correctness.

---

\*This work was supported in part by the Paul Ivanir Center for Robotics and Production Management at Ben-Gurion University of the Negev.

# 1 Introduction

The Entity Relationship (ER) data model was introduced by Chen [8] 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 framework for conceptual system and database design, and received many extensions and variations, which are generally termed the Enhanced ER (EER) model. The static object model of the Unified Modeling Language (UML) [2, 3] is essentially an extension of the EER model.

Traditionally, the EER model was used for conceptual database design. Used in this way, an EER schema is translated into logical database schemas, usually relational or Object-Oriented (OO) schemas ([17, 10, 9, 11]). Application program transactions and consistency checks are added directly to the target database schema (there are also few suggestions to extend the ER models with rules [26, 24]). Nowadays, with the inclusion of the EER model within the UML rising standard for object-oriented analysis and design, EER is used also for conceptual modeling of real time systems. The evolution of Internet applications and semi-structured databases, gives rise to a new direction, where EER serves as a conceptual model for the design of Web applications (e.g., [13, 14] and semi-structured schemas [20]).

An EER schema supports the specification of *cardinality constraints*, which restrict the dependencies among entities within a relationship type (see, for example, [22, 21, 12, 27]). For example, cardinality constraints can specify that a department must have at least five workers and at most eighty, or that for every combination of a course taught by some professor, there must be at least one text book, and no more than three text books. 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 logical database schema, or implemented within a system, 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 system. Moreover, at this level it is up to application programmers to correctly capture the constraints.

Lazarevic and Misic [18], suggest an extension of the ER model with structural integrity constraints: Entity and relationship types are extended with constraints concerning actions that can or should be associated with different update operations. The actions specify whether an update can

be restricted, or should invoke further updates in order to preserve cardinality integrity. The idea is that the constraints can be used for algorithmic design of integrity preserving update procedures, to be utilized in database application programs.

Bouzeghoub and Metais ([4, 5]) introduce a semantic modeling methodology, that enables the specification of various integrity constraints (including cardinality constraints and functional dependencies) and behavioral rules. The behavioral rules enable the inclusion of actions in their conclusion part, and therefore, can be used for specification of integrity enforcement operations on the semantic level. The semantic schema is mapped to an Object-Oriented database schema (specifically, the  $O_2$  data model [19]). The integrity constraints and the behavior rules are transformed into *constraint methods* that perform the integrity enforcement.

In this paper we further develop these ideas, and apply them to cardinality constraints. We extend the EER data model with *structure methods*, i.e., update methods that take the cardinality constraints into account. The new model is called MEER (Methods enhancement of EER). The structure methods are provably *cardinality faithful* and *terminating*. That is, they do not insert new inconsistencies in the database, can only decrease existing ones, and when the schema is strongly satisfiable, their applications include terminating branches. A refined approach towards measuring the cardinality consistency of an EER database instance is also introduced. The termination proof required an extension of earlier results of [21] about strong satisfiability of ER schemas.

The structure 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 MEER into some target schema language, in terms of the primitive methods alone. We plan to use this property in extending the translation of an EER schema into OO or relational schemas to account also for the structure methods.

The contribution of this paper is in the automatic creation of structure methods on top of a given EER schema, and in building the formal basis for proving their correctness. Our work can also be classified under the consistency enforcement direction. It closely relates to integrity enforcement in the active database paradigm ([7, 29]). The main difference is that while in the Event-Condition-Action approach consistency is enforced by propagation of actions that are triggered by events, a single structure method is responsible for the complete enforcement of the violated constraints. In that sense, the MEER approach is more holistic.

In Section 2 the EER model is formally defined, in a way that enables further extension with

methods. In Section 3 the MEER extension of EER is introduced, and proved to preserve database integrity. Section 4 is the conclusion. Proofs are postponed to the appendix.

## 2 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*. 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 "participants" related by a relationship type is its *arity* ( $\geq 2$ ). The role that an entity type plays within a relationship type is specified by a *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*. More general models allow also *high order relationship types*, among other relationship types ([25, 28]). We do not consider such relationship types, below.

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 terminology and notation with different semantics ([22, 21, 12, 27]). Below, we describe the most frequent cardinality constraints.

A key of a type is a means for identifying the instances of the type via their attributes. A set of attributes used for identification is called a *key* for the type. Most entity types, marked as *strong*, have at least one *mandatory key*. A relationship type might have an *optional key*. An entity type that is not strong is marked as *weak*. The entities of a weak entity type are identified by their inter-relationships to other entities. That is, a key of a weak entity type is defined by optional *partial key* attributes, and through *owner* entity types, that are related through *identifying relationship types*. For further details concerning weak entity types consult [28, 1].

An entity type symbol may be associated, as a *super-type*, with a set of entity type symbols (other than itself) that form its *specialization* or *sub-typing*. The sets of attributes of an entity type and of any of its super-types are disjoint (no over-writing). The specialization can specify that the sub-types are disjoint, or that they cover the whole super-type. An entity type symbol may participate, as a subtype, in at most a single specialization. The restriction of sub-typing to single inheritance is characteristic to most ER models. However, the integrity methods introduced in this paper stay valid also in the presence of multiple inheritance.

## 2.1 EER – a Formal Definition

An EER schema consists of *Entity Type* symbols, *Relationship Type* symbols (each with associated arity), *Role Name* symbols, *Attribute* symbols, and their inter-associations<sup>1</sup>. That is, an EER schema  $\mathcal{ER}$  is a quadruple of symbol sets and a quadruple of associations, as follows:

1. **Symbols:**  $(\mathcal{E}, \mathcal{R}, \mathcal{RN}, \mathcal{A})$ , where  $\mathcal{E}$  is the set of entity type symbols, already marked as *strong* or *weak*.  $\mathcal{R}$  is the set of relationship type symbols,  $\mathcal{RN}$  is the set of role name symbols, and  $\mathcal{A}$  is the set of attribute symbols, already marked as *simple* or *composite* attributes, as *single* or *multiple-valued* symbols, and with associations of composite attributes with other attributes.
2. **Associations:**  $(att, rel, sub, CC)$ , where
  - $att : \mathcal{E} \cup \mathcal{R} \rightarrow \mathcal{P}(\mathcal{A})$ , a partial mapping, which is mandatory for strong entity type symbols. It includes marking of *keys*.
  - $rel : \mathcal{R} \rightarrow \mathcal{P}(\mathcal{RN} \times \mathcal{E})$ . It includes the marking of identifying relationship types for weak entity types. The number of elements in  $rel(R)$  is the *arity* of  $R$ .
  - $sub$  is an irreflexive, acyclic ordering on  $\mathcal{E}$ . It induces the partial *sub-typing* mapping  $subtyping : \mathcal{E} \rightarrow \mathcal{P}(\mathcal{E})$ . The *subtyping* mapping satisfies:  $subtyping(E_1) \cap subtyping(E_2) = \emptyset$ . The sub-typing mapping can be extended to account for marking of disjointness and of covering of the super-type.
  - $CC$  is a mapping from  $\mathcal{R}$  to a set of cardinality constraints, described below.

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. Cardinality constraints are denoted by number pairs on the lines connecting diamonds and rectangles.

A *database instance*  $\mathcal{D}$  of an EER schema  $\mathcal{ER}$  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  $A$

---

<sup>1</sup>Our formulation follows, to some extent, that of [6].

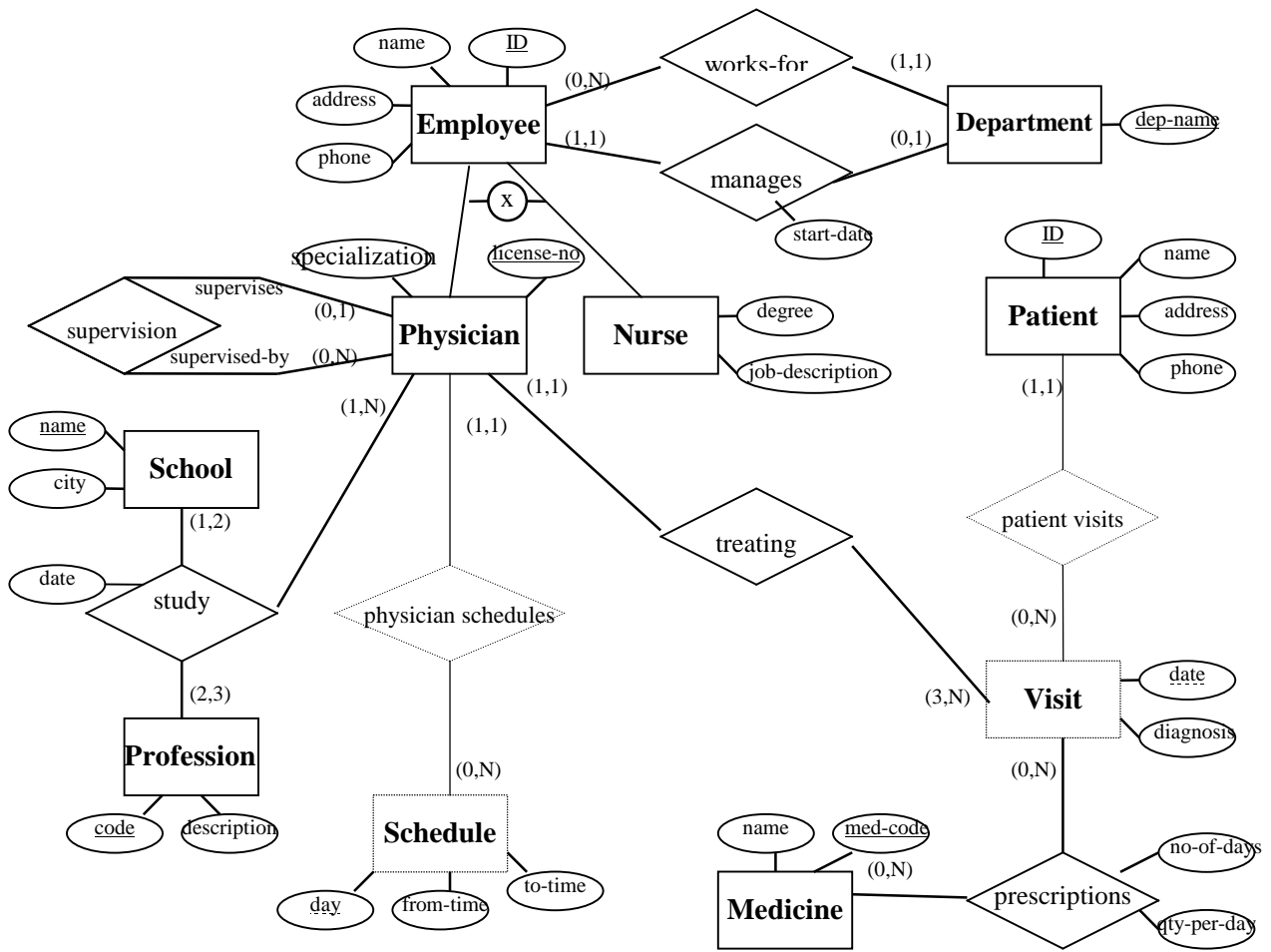


Figure 1: An EER diagram for a medical clinic information system

and a type symbol  $T$  (entity or relationship) with a value domain  $dom(A, T)$ .  $dom$  is extended to composite attributes by defining  $dom(A, T) = dom(A_1, T) \times \dots \times dom(A_n, T)$ , for a composite attribute symbol  $A$  that is associated with the attribute symbols  $A_1, \dots, A_n$ . 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  $\mathcal{ER}$  is denoted  $s^{\mathcal{D}}$ . It is defined as follows:

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

## 2.2 Cardinality Constraints

A cardinality constraint in  $CC(R)$  is set on a relationship type  $R^{\mathcal{D}}$ . It sets a minimum and a maximum restrictions on the cardinality dependency between two disjoint sets of roles of  $R^{\mathcal{D}}$ ,  $\{RN_{i_1}, \dots, RN_{i_k}\}$  and  $\{RN_{j_1}, \dots, RN_{j_l}\}$ <sup>2</sup>. In terms of relational algebra operations, a cardinality constraint is defined as follows:

$$CC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max](e_1, \dots, e_k) =$$

$$min \leq \mathbf{cardinality}(\sigma_{RN_{i_1}=e_1, \dots, RN_{i_k}=e_k}(\mathit{Project}_{RN_{i_1}, \dots, RN_{i_k}, RN_{j_1}, \dots, RN_{j_l}}(R^{\mathcal{D}}))) \leq max$$

where  $e_1, \dots, e_k$  are entities in the entity types  $E_{i_1}^{\mathcal{D}}, \dots, E_{i_k}^{\mathcal{D}}$ , that correspond to the roles  $RN_{i_1}, \dots, RN_{i_k}$ , respectively. That is,  $CC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max]$ , states the minimum and maximum number of  $RN_{j_1}, \dots, RN_{j_l}$  entity combinations, to which a given  $RN_{i_1}, \dots, RN_{i_k}$  entity tuple can be related within  $R^{\mathcal{D}}$ .

---

<sup>2</sup>Our characterization is based, to a great extent, on [27].

A cardinality constraint can be set either on the participating entity types, or on the relationship type itself. In the first case, termed an *Entity cardinality constraint*, the constraint is set on all entity tuples from  $E_{i_1}^{\mathcal{D}}, \dots, E_{i_k}^{\mathcal{D}}$ , while in the second case, termed a *Relationship cardinality constraint*, the constraint is set on all selections of entity tuples  $e_{i_1}, \dots, e_{i_k}$  from relationships in  $R^{\mathcal{D}}$ . For example, for a relationship type that relates sales persons, products, cities, and year-periods, an entity cardinality constraint might state that every sales person must sell every product in at least 3 and at most 5 City-Year-period combinations. A relationship cardinality constraint might state that if a sales person sells a product, then she must sell it in at least 3 and at most 5 City-Year-period combinations.

1. **Entity cardinality constraints:**

$$ECC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max] =$$

$$\forall (e_1, \dots, e_k) \in E_{i_1}^{\mathcal{D}}, \dots, E_{i_k}^{\mathcal{D}} : CC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max](e_1, \dots, e_k)$$

2. **Relationship cardinality constraints:**

$$RCC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max] =$$

$$\forall \{RN_1 : e_1, \dots, RN_n : e_n\} \in R^{\mathcal{D}} : CC_{\{RN_{i_1}, \dots, RN_{i_k}\}, \{RN_{j_1}, \dots, RN_{j_l}\}}[min, max](e_{i_1}, \dots, e_{i_k})$$

The more practical cases of cardinality constraints are when  $k + l = n$ , where  $n$  is the arity of the relationship type symbol  $R$ . In these cases, the definition of cardinality constraints is simplified as follows:

$$CC_{RN_{i_1}, \dots, RN_{i_k}}[min, max](e_1, \dots, e_k) = min \leq \mathbf{cardinality}(\sigma_{RN_{i_1}=e_1, \dots, RN_{i_k}=e_k}(R^{\mathcal{D}})) \leq max.$$

where  $e_1, \dots, e_k$  are entities in the entity types  $E_{i_1}^{\mathcal{D}}, \dots, E_{i_k}^{\mathcal{D}}$ , that correspond to the role names  $RN_{i_1}, \dots, RN_{i_k}$ , respectively.

Two special cases are when  $k = 1$ , and when  $k = n - 1$ . The first case is, sometimes, called a *participation cardinality constraint*, and the second case is, sometimes, called a *look-across cardinality constraint*. A participation cardinality constraint restricts the minimum and maximum number of tuples to which an entity can be related within  $R^{\mathcal{D}}$ . A look across cardinality constraint restricts the minimum and maximum number of entities, to which a combination of  $n - 1$  entities can be related within  $R^{\mathcal{D}}$ . Both, participation and look-across constraints can be set on entities or on relationships. All in all there are four special cases:

1. **Entity participation cardinality constraints:**  $ECC_{RN_{i_1}}[min, max]$ .
2. **Entity look-across cardinality constraints:**  $ECC_{RN_{i_1}, \dots, RN_{i_{n-1}}}[min, max]$ .
3. **Relationship participation cardinality constraints:**  $RCC_{RN_{i_1}}[min, max]$ .
4. **Relationship look-across cardinality constraints:**  $RCC_{RN_{i_1}, \dots, RN_{i_{n-1}}}[min, max]$ .

It is easy to show that for binary relationship types, participation and look-across, of either Entity or Relationship kind constraints, are essentially the same (see, for example, [12, 27]). However, for non-binary relationship types, participation and look-across express essentially different restrictions.

For the purpose of enriching an EER schema with structure methods we decided to restrict the EER schema to allow only the most frequent kinds of cardinality constraints: For binary relationship types – only Entity look-across cardinality constraints  $ECC_{RN_{i_1}}[min, max]$ , and for non-binary relationship types – only Relationship look-across cardinality constraints  $RCC_{RN_{i_1}, \dots, RN_{i_{n-1}}}[min, max]$ . We selected these constraints since we believe that they are the most relevant. Of course, the MEER model can be extended to account for other types of cardinality constraints.

Cardinality constraints are visualized as  $(min, max)$  labels on the lines that connect entity type rectangles with relationship type diamonds. Look-across constraints, that are set on  $n - 1$  roles out of  $n$  roles in a relationship, are visualized as a  $(min, max)$  label on the line associated with the role that is not included in the constraint. Therefore, for a binary relationship type  $R$  with  $rel(R) = \{(RN_1, E_1), (RN_2, E_2)\}$  and  $CC(R) = \{ECC_{RN_1}[min_1, max_1], ECC_{RN_2}[min_2, max_2]\}$ ,  $ECC_{RN_1}[min_1, max_1]$  that restricts the number of  $E_2$  entities that can be related to a given  $E_1$  entity is visualized on the line associated with the  $RN_2$  role between  $E_2$  and  $R$ , and  $ECC_{RN_2}[min_2, max_2]$  is visualized on the line between  $E_1$  and  $R$ . Similarly, for an  $n$ -ary relationship type  $R$ , a  $(min_k, max_k)$  label on the line associated with the role  $RN_k$  role between  $E_k$  and  $R$ , represents the constraint  $RCC_{RN_{i_1}, \dots, RN_{i_{n-1}}}[min_k, max_k]$ , where  $RN_k$  is the role not included within the roles  $RN_{i_j}$ , ( $1 \leq j \leq n - 1$ ). In Figure 1,

$CC(\mathbf{supervision}) = \{ECC_{\mathbf{supervises}}[0, \infty], ECC_{\mathbf{supervised-by}}[0, 1]\}$ . Hence, the **supervises** line is labeled  $(0, 1)$ , and the **supervised-by** line is labeled  $(0, \infty)$ . Similarly,

$CC(\mathbf{study}) = \{RCC_{\mathbf{study-School, study-Profession}}[1, \infty],$   
 $RCC_{\mathbf{study-School, study-Physician}}[2, 3],$   
 $RCC_{\mathbf{study-Physician, study-Profession}}[1, 2]\}$ .

A conventional simplified cardinality notation for binary relationships 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. In the sequel, we summarize the information that a schema associates with a relationship type  $R$ , as a relationship construct  $R(RN_1 : E_1[min_1, max_1], \dots, RN_n : E_n[min_n, max_n])$ . For example, the **study** relationship construct in Figure 1 is

**study**( **study-School** : **School**[1, 2],  
**study-Profession** : **Profession**[2, 3],  
**study-Physician** : **Physician**[1,  $\infty$ ] ).

### 2.3 Consistency of a Database Instance

A database instance of a schema  $\mathcal{ER}$  is *consistent* if it satisfies the intended meaning of keys, relationship constructs, cardinality constraints, and sub-typing relationships. An EER schema is *consistent (satisfiable)* if it has a consistent database instance. The constraints set by keys and sub-typing relationships were already described above. Note that the latter imply inheritance of attributes and relationships through specialization relationships. A relationship construct  $R(RN_1 : E_1[min_1, max_1], \dots, RN_n : E_n[min_n, max_n])$  in  $\mathcal{ER}$  imposes the following constraints:

1.  $R^{\mathcal{D}} \subseteq E_1^{\mathcal{D}} \times \dots \times E_n^{\mathcal{D}}$ .
2. The cardinality bounds on the  $i$ -th component specify the minimum and maximum restrictions that the look-across cardinality constraint sets on the rest of the  $n - 1$  components. As already stated, for binary relationship types the constraint is an Entity cardinality constraint, while for non-binary relationship types the constraint is a Relationship cardinality constraint. That is:
  - (a) Binary relationship types: For  $i = 1, 2$  and  $j = 2, 1$ , respectively,  $ECC_{RN_i}[min_j, max_j]$  must hold.
  - (b) Non-binary relationship types: For  $i = 1, \dots, n$ ,  $RCC_{RN_1, \dots, RN_{i-1}, RN_{i+1}, \dots, RN_n}[min_i, max_i]$  must hold.

Two more characterizations of database instances should be introduced. The first is the notion of *strong satisfiability* of an EER schema, introduced in [21]. An EER schema is strongly satisfiable if for every entity or relationship type symbol  $T$  there exists a consistent database instance  $\mathcal{D}$  such that  $T^{\mathcal{D}}$  is not empty. Strong satisfiability of a schema means that the cardinality constraints

associated with each type symbol in the schema are not essentially contradictory, since each type symbol can have a non-empty extension within a consistent database instance. In [21] it is shown that a strongly satisfiable schema has a consistent instance with non-empty extensions for all type symbols. Moreover, two characterizations of strong satisfiability are provided: First, in terms of solutions of a linear inequality system, and second in terms of critical cycles in a graph associated with a schema. The EER model discussed in [21] includes only entity participation cardinality constraints, and allows no sub-typing. The results can be extended as described in Propositions 2.1 and 2.3 below. Both propositions can be proved by showing correspondence of database instances.

**Proposition 2.1** *Let  $\mathcal{ER}$  be an EER schema that includes only entity participation cardinality constraints and sub-typing. Let  $\mathcal{ER}'$  be an EER schema that is obtained from  $\mathcal{ER}$  by replacing every sub-typing relationship by a regular relationship with  $(1, 1)$  participation for the subtype and  $(0, 1)$  participation for the super-type. Then,  $\mathcal{ER}$  is strongly satisfiable iff the transformed schema  $\mathcal{ER}'$  is so.*

For the purpose of handling strong satisfiability of a schema with look-across cardinality constraints in non-binary relationship types, we first define a transformation  $\tau$  that removes all such constraints.

**Definition 2.2** *Let  $\mathcal{ER}$  be an EER schema that includes, for non-binary relationship types, relationship look-across cardinality constraints. Then  $\tau(\mathcal{ER})$  is a schema without such constraints. It is obtained from  $\mathcal{ER}$  by applying the following transformation to every non-binary relationship type symbol  $R$  with a relationship construct  $R(RN_1 : E_1[\min_1, \max_1], \dots, RN_n : E_n[\min_n, \max_n])$  (implying  $n$  look-across cardinality constraints  $RCC_{RN_{i_1}, \dots, RN_{i_{n-1}}}[\min, \max]$ , where  $1 \leq i_1, \dots, i_{n-1} \leq n$ ):*

1. *Remove all relationship look-across cardinality constraints on  $R$ .*
2. *Insert  $n$  new entity type symbols  $ER_1, \dots, ER_n$ . The intension is that in a database instance  $\mathcal{D}$ , the entity type  $ER_i^{\mathcal{D}}$  represents the projection of  $R^{\mathcal{D}}$  on all of its roles apart from the  $i$ -th role  $RN_i$ .*
3. *Connect each  $ER_i$  to  $E_1, \dots, E_n$  by new binary relationship type symbols  $R_{i,1}, \dots, R_{i,n}$ , with the relationship constructs:*

- For  $1 \leq i, j \leq n, i \neq j$ :  $R_{i,j}(ERN_i : ER_i[0, \infty], RN_j : E_j[1, 1])$ . That is, each  $ER_i^{\mathcal{D}}$  entity relates to exactly one entity of  $E_j^{\mathcal{D}}$ .
- For  $1 \leq i \leq n$ :  $R_{i,i}(ERN_i : ER_i[0, \infty], RN_i : E_i[\min_i, \max_i])$ . That is, the number of  $E_i^{\mathcal{D}}$  entities that can be related to a single  $ER_i^{\mathcal{D}}$  entity is bounded by  $\min_i$  and  $\max_i$ .

These constraints restrict an  $ER_i^{\mathcal{D}}$  entity to be related to a single  $n - 1$  ary tuple from  $E_1^{\mathcal{D}}, \dots, E_{i-1}^{\mathcal{D}}, E_{i+1}^{\mathcal{D}}, \dots, E_n^{\mathcal{D}}$ .

**Proposition 2.3** *Let  $\mathcal{ER}$  be an EER schema that includes, for non-binary relationship types, relationship look-across cardinality constraints. Then,  $\mathcal{ER}$  is strongly satisfiable iff the transformed schema  $\tau(\mathcal{ER})$ , together with the following constraints, is:*

1. For  $1 \leq i \leq n$ , let  $\mathcal{R}_i = R_{i,1} \bowtie_{ERN_i} R_{i,2} \dots \bowtie_{ERN_i} R_{i,n}$ . Then all  $\pi_{RN_1, \dots, RN_n}(\mathcal{R}_i)$ , for  $i = 1, \dots, n$ , are equal. That is, the original  $R^{\mathcal{D}}$  can be reconstructed from the newly inserted types.
2. For each  $i$ , the relation  $\mathcal{R}_i$  satisfies the functional dependency:  $RN_1 \dots RN_{i-1} RN_{i+1} \dots RN_n \rightarrow ERN_i$ . This constraint enforces the intension behind  $ER_i$ , as explained in the definition of  $\tau$ .

These two propositions enable us to transform a MEER schema  $\mathcal{ER}$  into a schema  $\mathcal{ER}'$  with binary relationship types alone, to which the results of [21] apply. If the resulting schema  $\mathcal{ER}'$  also satisfies the conditions set in Proposition 2.3, then the two schemas are either both strongly satisfiable, or both not. We use this property later on, for proving that the integrity methods of MEER have terminating branches for strongly satisfiable EER schemas (the methods are non-deterministic).

Another characterization refers to the extensions of entity type symbols. A consistent database instance is called *proper* if the extensions of entity type symbols that are not related through specialization/generalization relationships, do not intersect. In a proper database instance, an entity belongs to at most a single entity type hierarchy chain. Hence, any update involving an entity  $e$ , should consider only the single hierarchy chain of entity types to which  $e$  belongs. The integrity methods of MEER are proved to be *cardinality faithful* (see below), only when applied to a proper database instance.

### 3 MEER – EER with Methods

The MEER data model extends the EER schema with *structure methods*, which are update methods that are sensitive to the cardinality constraints. That is, applying a structure method to a consistent database instance of an EER schema, preserves the consistency. The structure methods are defined on top of *primitive update methods*, that are integrity insensitive.

#### Terminology and Notation

The addition of methods requires specification of the information available in an EER schema, and associated with entities of its database instances. To improve intuition, we use the conventional object-oriented “dot notation” for selecting the information associated with a syntactic element.

#### Schema level terminology:

1. For an entity type symbol  $E$ , the super types and the subtypes of  $E$  (direct and indirect) are given by  $E.supers$  and  $E.subs$  (NULL if none). The relationship type symbols whose constructs involve  $E$  or an entity type in  $E.supers$ , and the corresponding role names, are given by  $E.rels = \{[R, RN] \mid RN(R) = E', E' = E \text{ or } E' \in E.supers\}$ .  $E.bin\_rels$  selects the  $[R, RN]$  pairs for the binary relationship type symbols,  $E.n\_rels$  selects the pairs for the non-binary relationship type symbols. The attributes and the keys of  $E$  (including those of its super-types) are given by  $E.attributes$  and  $E.keys$ , respectively.
2. For a relationship type symbol  $R$ , the arity is  $R.arity$ , the attribute symbols are  $R.attributes$ , and the role names are given by  $R.role\_names$ . The entity type symbol  $RN(R)$  identified by the role name  $RN$ , is denoted  $R.E\_of\_RN$ ;  $RN(R)$  and all of its subtypes, is denoted  $R.Es\_of\_RN$ .  $R.min\_of\_RN$  and  $R.max\_of\_RN$  are the minimum and maximum cardinalities for  $RN$ .

**Database level terminology:** Let  $\mathcal{D}$  be a database instance of a schema  $\mathcal{ER}$ .

1. For an entity  $e$  of an entity type  $E^{\mathcal{D}}$ ,  $e.A$  retrieves the value on  $e$  of attribute  $(A, E')^{\mathcal{D}}$ , where  $E'$  is either  $E$  or a super type of  $E$  (i.e.,  $E' \in \{E\} \cup E.supers$ ). Note that  $e.A$  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  $\mathcal{D}$  is a value in the domain that  $\mathcal{D}$  assigns to a key attribute symbol  $A$  of  $E$ .

For every  $[R, RN]$  in  $E.rels$ ,  $e.rels([R, RN])$  denotes the set of  $R$  relationships whose  $RN$  component is  $e$ , and  $e.no\_of\_rels([R, RN])$  is its cardinality. Recall that a relationship is a labeled set:  $\{RN_1 : e_1, \dots, RN_n : e_n\}$ .

2. For a relationship  $\vec{r}$  of a relationship type  $R^{\mathcal{D}}$ ,  $\vec{r}.A$  retrieves the value of attribute  $(A, R)^{\mathcal{D}}$  on  $\vec{r}$ .  $\vec{r}.RN$  retrieves the  $RN$  entity component of  $\vec{r}$ , for every role name  $RN \in R.role\_names$ .

A *legal relationship* for  $R$  in  $\mathcal{D}$  is a labeled set  $\{RN_1 : e_1, \dots, RN_n : e_n\}$ , such that  $R.role\_names = \{RN_1, \dots, RN_n\}$ , and  $e_i$  is an entity in an entity type identified by the role name  $RN_i$ , i.e.,  $e_i \in (E')^{\mathcal{D}}$ , for  $E' \in R.Es\_of\_RN_i$ .

### 3.1 Primitive methods

The primitive methods perform *insertion*, *deletion*, *retrieval*. in a database instance  $\mathcal{D}$  of an EER schema  $\mathcal{ER}$ . They are associated with the entity and the relationship type symbols of the schema. Insertions involve the creation of a new entity or relationship. Consequently, a database instance created only with the primitive methods is proper.

The primitive methods are sensitive to the subtype and super-type relations, in the sense that an insertion of an entity to the entity type  $E^{\mathcal{D}}$  inserts it also to all super entity types of  $E^{\mathcal{D}}$ , and a deletion of an entity from  $E^{\mathcal{D}}$  deletes it also from all sub entity types of  $E^{\mathcal{D}}$ . Attribute modification is not handled in this paper since it has no relevance to the cardinality constraints.

#### 1. Primitive methods for an entity type symbol $E$ :

Let  $v$  be a legal key value for  $A$  of  $E$  in  $\mathcal{D}$  ( $v$  is either a tuple  $(v_1, \dots, v_k)$ , if  $A$  is composite, or atomic, otherwise).

- (a)  $E.insert(A : v)$  – Creates a new  $E^{\mathcal{D}}$  entity  $e$ , such that  $e.A = v$ , and for every  $[R, RN]$  in  $E.rels$ ,  $e.no\_of\_rels([R, RN]) = 0$ , and  $e.rels([R, RN]) = \emptyset$ . The entity  $e$  is added to all entity types  $E'^{\mathcal{D}}$  such that  $E' \in E.supers$ . If there is already an  $E^{\mathcal{D}}$  entity  $e$  identified by the key value  $A : v$  the method does nothing. The return value is  $e$ .
- (b)  $E.delete(A : v)$  – Deletes from  $E^{\mathcal{D}}$  and from all entity types  $E'^{\mathcal{D}}$  such that  $E' \in E.subs$ , the entity  $e$  identified by the value  $v$  of  $(A, E)^{\mathcal{D}}$  ( $e.A = v$ ), if any. The return value is  $e$ .

- (c)  $E.\mathbf{retrieve}(A : v)$  – Retrieves from  $E^{\mathcal{D}}$  the entity  $e$  identified by  $e.A = v$ , if any. The return value is either  $e$ , or NULL, if there is no such entity.  $E.\mathbf{retrieve}$  can accept also an entity ID.
- (d)  $E.\mathbf{retrieve\_all}()$  – Retrieves all entities in  $E^{\mathcal{D}}$ .

## 2. Primitive methods for a relationship type symbol $R$ :

Let  $\vec{r} = \{ RN_1 : e_1, \dots, RN_n : e_n \}$  be a legal relationship for  $R$  in  $\mathcal{D}$ .

- (a)  $R.\mathbf{insert}(\vec{r})$  – Creates an  $R^{\mathcal{D}}$  relationship  $\vec{r}$  with  $\vec{r}.RN_i = e_i$  ( $1 \leq i \leq n$ ). The return value is  $\vec{r}$ . The entities  $e_i$  ( $1 \leq i \leq n$ ) are updated as follows:  $e_i.no\_of\_rels([R, RN_i])$  is increased by one, and  $\vec{r}$  is added to  $e_i.rels([R, RN_i])$ .
- (b)  $R.\mathbf{delete}(\vec{r})$  – Deletes from  $R^{\mathcal{D}}$  the specified relationship, if any. If there is a deletion, the entities  $e_i$  ( $1 \leq i \leq n$ ) are updated to decrease  $e_i.no\_of\_rels([R, RN_i])$  by one, and remove  $\vec{r}$  from  $e_i.rels([R, RN_i])$ .
- (c)  $R.\mathbf{retrieve}(\vec{r})$  – Retrieves  $\vec{r}$  from  $R^{\mathcal{D}}$ , if any. The return value is either  $\vec{r}$ , or NULL, if there is no such relationship.
- (d)  $R.\mathbf{retrieve\_all}()$  – Retrieves all relationships in  $R^{\mathcal{D}}$ .

## 3.2 Properties of Update Methods: Consistency and Termination

The consistency situation of a database instance is, usually, evaluated on a true-false scale. A more relaxed approach might enable evaluation on a richer scale. Under such an approach, the consistency situation of a database instance can be quantified over some consistency scale, and the consistency situation of different database instances can be compared (resulting in a partial ordering of database instances). An updated database instance may become *more consistent* or *less consistent*, if its quantified consistency measure grows or decreases, respectively. This approach can be useful, for example, when dealing with legacy systems that were generated when the integrity constraints were not known, and the DBMS might not wish to enforce new constraints on them.

An update method is *integrity faithful* if its application to a database instance does not make it less consistent. Clearly, integrity faithful update methods preserve consistency. An update method is *consistency sensitive* if it enforces only as much consistency as needed to be integrity faithful. That is, the updates that it invokes are justified by the need to be integrity faithful.

The consistency of a database instance of an EER schema can be measured by counting the deviations of its entities and relationships from their minimum and maximum requirements, and counting the references to non-existing entities (i.e., entities that have been removed from the database). The integrity methods in MEER take into account the structure constraints that the EER schema imposes on its database instances. Being integrity faithful in this context means that newly inserted entities and relationships do not insert new inconsistencies, deletions do not insert new inconsistencies or cause references to non existing entities, and the consistency status of existing entities and relationships can only improve. This is summarized in the definition of a *cardinality faithful* update method below (Definition 3.4). The integrity methods in MEER are provably cardinality faithful. They are designed towards being also consistency sensitive, in the sense that all updates invoked by an integrity method are justified by the cardinality faithfulness requirement. This is important since propagation of updates can be quite expensive.

### 3.2.1 Cardinality Faithfulness

In order to formally define *cardinality faithfulness* we need to define measures for consistency and inconsistency of entities and relationships:

**I. Consistency of an entity:** This is measured by the properties *min\_inconsistency*, *max\_inconsistency*, and *consistency* of an entity with respect to an entity type. These measures take into account only cardinality constraints set on entity types, since cardinality constraints set on relationship types are handled by consistency measures of relationships. Consequently, in MEER, these properties involve only binary relationship constructs.

#### Definition 3.1 Inconsistency terms for entities:

Let  $\mathcal{D}$  be a database instance of an EER schema,  $E$  an entity type symbol in the schema, and  $e \in E^{\mathcal{D}}$  an entity:

1. For  $[R, RN] \in E.bin\_rels$ , where  $R.role\_names = \{RN, RN'\}$ , define:

$$\begin{aligned} min\_inconsistency_E(e)([R, RN]) &= \begin{cases} R.min\_of\_RN' - e.no\_of\_rels([R, RN]) & \text{if positive} \\ 0 & \text{otherwise} \end{cases} \\ max\_inconsistency_E(e)([R, RN]) &= \begin{cases} e.no\_of\_rels([R, RN]) - R.max\_of\_RN' & \text{if positive} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

$$2. \min\_inconsistency_E(e) = \sum \min\_inconsistency_E(e)([R, RN])$$

$$\max\_inconsistency_E(e) = \sum \max\_inconsistency_E(e)([R, RN])$$

where the sums are taken over all  $[R, RN] \in E.bin\_rels$ . An entity becomes *more consistent* when its *min\_inconsistency* or its *max\_inconsistency* decrease.

$$3. \text{consistent}_E(e) \equiv \min\_inconsistency_E(e) = \max\_inconsistency_E(e) = 0.$$

For example, if in a database instance of the schema described in Figure 1 a **Physician** entity  $p$  is not related by any relationship, then:

$$\min\_inconsistency_{\mathbf{Physician}}(p)([\mathbf{treating}, \mathbf{treating.Physician}]) = 1;$$

$$\min\_inconsistency_{\mathbf{Physician}}(p)([\mathbf{works-for}, \mathbf{works-for.Employee}]) = 1;$$

where **treating.Physician** and **works-for.Employee** are schema provided role names.

$\min\_inconsistency_{\mathbf{Physician}}(p)$  with respect to the two other binary relationship types that involve **Physician**, i.e., **physician-schedules** and **supervision**, is 0. Consequently,

$$\min\_inconsistency_{\mathbf{Physician}}(p) = 2, \text{ and } p \text{ is inconsistent.}$$

Null entities (introduced below) are consistent by definition. That is, for a null entity  $null_e$ :

$$\min\_inconsistency_E(null_e) = \max\_inconsistency_E(null_e) = 0$$

Note that, if  $\min\_inconsistency_E(e)([R, RN]) \neq 0$  then  $\max\_inconsistency_E(e)([R, RN]) = 0$ , and vice versa.

**II. References to a non-existing entity:** This is measured by the properties *referential\_inconsistency*, and *referential\_consistency* of an entity with respect to an entity type:

**Definition 3.2 Referential inconsistency:**

Let  $\mathcal{D}$  be a database instance of an EER schema,  $E$  an entity type symbol in the schema, and  $e$  an entity in the domain of  $\mathcal{D}$ :

$$\text{referential\_inconsistency}_E(e) = \begin{cases} 0 & e \in E^{\mathcal{D}} \\ \sum e.no\_of\_rels([R, RN]) \text{ for all } [R, RN] \in E.rels & \text{otherwise} \end{cases}$$

An entity becomes *less referentially inconsistent* with respect to  $E$  when its *referential\_inconsistency* $_E$  decreases. An entity  $e$  is *referentially consistent* with respect to  $E$  if *referential\_inconsistency* $_E(e) = 0$ , and is *referentially inconsistent* with respect to  $E$ , otherwise.

**III. Consistency of a relationship:** This is measured by the properties *min\_inconsistency*,

*max\_inconsistency*, and *consistency* of a relationship with respect to a relationship type. These measures are relevant only with respect to cardinality constraints set on relationship types, since cardinality constraints set on entity types, are handled by similar consistency measures of entities. Consequently, in MEER, these properties involve only non-binary relationship constructs.

**Definition 3.3 Inconsistency terms for relationships:**

Let  $\mathcal{D}$  be a database instance of an EER schema,  $R$  a non-binary relationship type symbol in the schema, and  $\vec{r} = \{ RN_1 : e_1, \dots, RN_n : e_n \} \in R^{\mathcal{D}}$  ( $n > 2$ ) a relationship:

1. For all  $i = 1 \dots n$ :

$$\begin{aligned} \min\_inconsistency_R(\vec{r})(i) &= \\ &\begin{cases} R.\min\_of\_RN_i - \mathbf{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)) & \text{if positive} \\ 0 & \text{otherwise} \end{cases} \\ \max\_inconsistency_R(\vec{r})(i) &= \\ &\begin{cases} \mathbf{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)) - R.\max\_of\_RN_i & \text{if positive} \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

2.  $\min\_inconsistency_R(\vec{r}) = \sum_{i=1}^n \min\_inconsistency_R(\vec{r})(i).$

$$\max\_inconsistency_R(\vec{r}) = \sum_{i=1}^n \max\_inconsistency_R(\vec{r})(i).$$

A relationship becomes *more consistent* when its *min\_inconsistency* or its *max\_inconsistency* decreases.

3.  $\mathit{consistent}_R(\vec{r}) \equiv \min\_inconsistency_R(\vec{r}) = \max\_inconsistency_R(\vec{r}) = 0.$

Binary relationships are consistent by definition (their minimum “relationship-type-restrictions” can be taken as 1, and their maximum “relationship-type-restrictions” can be taken as  $\infty$ ). That is, for a binary  $R$ , for every relationship  $\vec{r} \in R^{\mathcal{D}}$ :

$$\min\_inconsistency_R(\vec{r}) = \max\_inconsistency_R(\vec{r}) = 0$$

Note that if  $\min\_inconsistency_R(\vec{r})(i) \neq 0$  then  $\max\_inconsistency_R(\vec{r})(i) = 0$  ( $1 \leq i \leq n$ ), and vice versa.

Finally, the cardinality faithfulness property of an update method can be defined, based on the consistency and inconsistency measures for entities and relationships.

**Definition 3.4 Cardinality faithfulness:**

An update method  $M$  is cardinality faithful if, when its application to an arbitrary database instance terminates, the following conditions hold:

1. Newly inserted entities and relationships are consistent.
2. Existing entities and relationships can become only more consistent: Their *min\_inconsistency* and their *max\_inconsistency* can only decrease.
3. Referential inconsistencies of existing entities can only decrease.
4. All entities deleted during its operation are referentially consistent with respect to the entity types from which they are deleted.

In the Appendix we prove that all integrity methods in MEER are cardinality faithful. Therefore, the application of an integrity method to any consistent proper database instance can yield only consistent database instances. The restriction to proper database instances is necessary in order to avoid references to non-existing entities.

**3.2.2 Update Propagation Policies and Termination**

In order to achieve cardinality faithfulness, an integrity update method might invoke associated update methods. This propagation of updates might be complex and cause non-termination, since it might lead to nested propagations. Cardinality faithfulness is essential since it guarantees that a nested update does not spoil the efforts of an update that invokes it, or of an already completed update. Yet, the integrity methods do not enforce full consistency on existing instances. For example, in the **reconnect** method (introduced in Section 3.3.2), if the entity that loses a relationship becomes less minimally consistent, then a new relationship is connected to it, but the entity is not enforced to be consistent. In that sense, the integrity methods are consistency sensitive.

The propagation dictated by a complex EER schema can be quite annoying. Moreover, it is possible that a user wishes to perform an update, but lacks all the necessary data to complete it as a cardinality faithful update. Therefore, it seems that a more pragmatic approach should enable cardinality faithful updates that are performed in parts. For that purpose we allow entity types to be populated by *null entities*, which are “fictional” entities, to which cardinality constraints do not apply – they are consistent by definition. Another means to relax the burden of update propagation

is to revise the schema. The decision between the various options is left to the user (e.g., database administrator). The options are summarized below:

1. **Propagate** – an insertion or deletion of an instance violates a cardinality constraint, and invokes appropriate deletion or insertion actions.
2. **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 into an entity type and connected to some other “real” entities, so that their integrity is preserved. A null entity can be replaced (at a later time) by a real one by reconnecting its related entities to a real entity.
3. **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. 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 cardinality constraints.
4. **Reject** – the update operation is refused. This is in some sense a brute force action for integrity preservation.

Nested update propagations raise the issue of non-termination. Since the integrity methods allow the user (or administrator) to choose a policy, they are non-deterministic. Moreover, even an update propagation policy is non-deterministic since the user can decide to use various entities to connect or disconnect to an entity under consideration. In particular, the user can always decide to insert a new entity or relationship as a compensation for a missing connection. The question of termination, then, should refer to the existence of terminating computation paths (terminating branches in the computation tree).

**Definition 3.5 Termination:**

*An update method  $M$  is terminating with respect to a database instance  $\mathcal{D}$  if any application of  $M$  to  $\mathcal{D}$  includes terminating branches. An update method is terminating if it is terminating with respect to any consistent database instance.*

**Definition 3.6 Correctness:**

An update method  $\mathbf{M}$  is correct if it is cardinality faithful and terminating.

In the Appendix we prove that for strongly satisfiable EER schemas, all integrity methods in MEER are correct. Therefore, given a strongly satisfiable EER schema, the application of an integrity method to any consistent proper database instance can always terminate with a consistent database instance.

**3.3 Integrity Methods****3.3.1 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 subsection 3.3.2).

**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  $\mathcal{D}$ , involves the following operations: If  $A : v$  does not identify an already existing entity in  $E^{\mathcal{D}}$ , a new entity with the key value  $v$  for  $A$  is inserted into  $E^{\mathcal{D}}$ , 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  $E^{\mathcal{D}}$  are not met by the new entity, then the user is asked to provide entities for the missing relationships. New not-null entities cause propagation.

$E.\text{integrity\_insert}(A : v) :$       $e = E.\text{insert}(A : v);$   
    $E.\text{make\_consistent}(e).$

$E.\text{make\_consistent}(e) :$

While there exist  $[R, RN] \in E.\text{bin\_rels}$ , with  $R.\text{role\_names} = \{RN, RN'\}$  such that

$min\_inconsistency_E(e)([R, RN]) > 0$  do:

Select an  $[R, RN]$ . Denote  $R.E\_of\_RN'$  by  $E'$ .

ASK USER TO CHOOSE ONE POLICY OUT OF **Reject**, **Nullify**, **Propagate** or **Schema revision**:

– **Reject**: Break the update.

– **Nullify**:  $null\_e = E'.insert(A : null)$ ;

$R.connect(\{RN : e, RN' : null\_e\})$ .

– **Propagate**: ASK USER FOR A LEGAL KEY VALUE  $A : v$  for  $E'$ .

$e' = E'.insert(A : v)$ ;

$R.connect(\{RN : e, RN' : e'\})$ ; If failed, repeat **Propagate**.

If  $e'$  is new:  $E'.make\_consistent(e')$ .

– **Schema revision**: Decrease  $R.min\_of\_RN'$ .

### 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. In order to leave the new **Physician** entity consistent, we need to observe all binary relationship type symbols whose constructs involve the entity type symbol **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 three visits, and must work for some department. So, we have to ask the user for candidate visits 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 visits must be new entities, 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 existing 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 (null entities that are not connected to real entities should be deleted by the DBMS).  $\square$

The **integrity\_insert** method guarantees the consistency of the inserted entity. It involves insertion of entities and of binary relationships alone, and does not involve any deletions.

**Claim 3.7** *For a strongly satisfiable EER schema the **integrity\_insert** method is correct. That is:*

1. *The **integrity\_insert** method is cardinality faithful. In particular, if  $E.\text{integrity\_insert}(A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $E$  is inserted to  $E^{\mathcal{D}}$ .*
2. *The **integrity\_insert** method is terminating. Moreover, **integrity\_insert** is terminating with respect to any database instance that does not violate maximum cardinality constraints.*

**Proof:** In Appendix A  $\square$

## 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  $\mathcal{D}$ , involves the following operations: If  $A : v$  indeed identifies an already existing entity in  $E^{\mathcal{D}}$ , 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  $R^{\mathcal{D}}$ , for  $R$  in  $E.\text{rels}$ . 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



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 increases the *min\_inconsistency* of the **Visit** entity, and must invoke deletion of that entity. In order to avoid or terminate the propagation, the user might decide to **Nullify** the **Physician** entity in the **treating** relationship of that visit.  $\square$

The **integrity\_delete** method guarantees the referential consistency of the deleted entity. That is, it guarantees that in a proper database instance, a deleted entity is not referenced, and no new inconsistencies are introduced. Note that if the database instance is not proper, referential consistency cannot be guaranteed without exhaustive check of the entire database.

**Claim 3.8** *For a strongly satisfiable EER schema the **integrity\_delete** method is correct. That is:*

1. *The **integrity\_delete** method is cardinality faithful. In particular, if  $E.\text{integrity\_delete}(A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $E$ , if it exists, is deleted from  $E^{\mathcal{D}}$ .*
2. *The **integrity\_delete** method is terminating. Moreover, **integrity\_delete** is terminating with respect to any database instance that does not violate maximum cardinality constraints.*

**Proof:** In Appendix A  $\square$

### 3.3.2 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

$\vec{r} = \{ RN_1 : e_1, \dots, RN_n : e_n \}$  for  $R$  in a database instance  $\mathcal{D}$ , involves the following operations:

If all entities  $e_i$  ( $1 \leq i \leq n$ ) already exist, and insertion of the new relationship does not violate maximum cardinality constraints set on  $R$ , the new relationship is inserted into  $R^{\mathcal{D}}$ , and made consistent.

$R.\mathbf{connect}(\vec{r}) =$

If  $\forall 1 \leq i \leq n : (R.E\_of\_RN_i).\mathbf{retrieve}(e_i)$  and

$\mathbf{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)) < R.max\_of\_RN_i$

then  $R.\mathbf{insert}(\vec{r}); R.\mathbf{make\_consistent}(\vec{r})$

else **Reject**;

$R.\mathbf{make\_consistent}(\vec{r}) =$

For  $i = 1, \dots, n$ :

While  $min\_inconsistency_R(\vec{r})(i) > 0$  do:

ASK USER TO CHOOSE ONE POLICY OUT OF **Reject**, **Nullify**, **Connect**, or

**Schema revision**:

– **Reject**: Break the update.

– **Nullify**:  $null\_e = (R.E\_of\_RN_i).\mathbf{insert}(A : null);$

$R.\mathbf{connect}(\vec{r}_{RN_i=null\_e});$

where  $\vec{r}_{RN_i=null_e}$  is  $\vec{r}$  with  $null_e$  substituted for the role name  $RN_i$ .

- **Connect**: ASK USER FOR A LEGAL KEY VALUE  $A : v$  for  $R.E\_of\_RN_i$ .  
 $e = (R.E\_of\_RN_i).insert(A : v)$ ;  
 $R.connect(\vec{r}_{RN_i=e})$ ; If failed, repeat **Connect**.  
 If  $e$  is new:  $(R.E\_of\_RN_i).make\_consistent(e)$
- **Schema revision**: Decrease  $R.min\_of\_RN_i$ .

### 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 into 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 into the **study** relationship type. Since **study** is non-binary,  $study.make\_consistent(study.physician : p, study.profession : f, study.school : s)$  is applied. Assume, for example, that there are no other **study** relationships with the physician  $p$  and the school  $s$ . Then  $min\_inconsistency_{study}(study.physician : p, study.profession : f, study.school : s)(2) = 1$ . Then the user is asked to choose among **Reject**, **Nullify**, **Connect** or **Schema revision**. If **Nullify** is chosen, then a new null entity is inserted into 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 is made consistent by invoking **Profession.make\\_consistent** on it.  $\square$

The **connect** method does not involve deletion of entities or of relationships. Also, since binary relationships are consistent by definition, their connection involves no propagation ( $R.\mathbf{make\_consistent}(\vec{r})$  terminates immediately). That is, for binary relationship types, connection does not trigger new insertions of entities or of relationships. Hence, it is easy to characterize the correctness of the **connect** method, as stated in the following claim:

**Claim 3.9** *For a strongly satisfiable EER schema the **connect** method is correct. That is:*

1. *The **connect** method is cardinality faithful. In particular, if  $R.\mathbf{connect}(\vec{r})$  terminates without rejection, the relationship  $\vec{r}$  is inserted into  $R^{\mathcal{D}}$ . Moreover,*
  - (a) ***connect** propagates no deletions of either entities or relationships. Consequently, for every component  $e_i$  of the relationship  $\vec{r}$ ,  $e_i.\mathit{no\_of\_rels}([R, RN_i])$  increases. If  $R$  is binary, then  $\mathit{min\_inconsistency}_{R.E\_of\_RN_i}(e_i)$  decreases while  $\mathit{max\_inconsistency}_{R.E\_of\_RN_i}(e_i)$  stays intact.*
  - (b) *If  $R$  is binary, **connect** does not lead to any update propagation. Its application does not invoke neither **integrity\\_insert** (and its auxiliary  $E.\mathbf{make\_consistent}$  method) nor **connect** itself.*
2. *The **connect** method is terminating. Moreover, **connect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.*

**Proof:** In Appendix A  $\square$

## II. The disconnect method

The **disconnect** method for a relationship type symbol  $R$ , and a legal relationship  $\vec{r} = \{ RN_1 : e_1, \dots, RN_n : e_n \}$  for  $R$  in a database instance  $\mathcal{D}$ , involves the following operations: If  $R$  is non-binary, or if it is binary and the deletion of the relationship does not violate minimum cardinality constraints set on the entity types participating in  $R$ , then the relationship is disconnected. The restrictions for binary relationship types are intended to save the need to invoke entity deletions, in order to be cardinality faithful. In any case, if entity deletion is necessary, then it is more reasonable to start with the deletion of the related entities and not with their relationship. If

$R$  is non-binary, the effect is propagated to other relationships in  $R^{\mathcal{D}}$ , if needed. This propagation might require reconnection or disconnection of other relationships in  $R^{\mathcal{D}}$ .

$R.\text{disconnect}(\vec{r}) =$

If  $R$  is non-binary, or

$R$  is binary and for  $i, j = 1, 2, i \neq j$ :

$e_i.\text{no\_of\_rels}([R, RN_i]) > R.\text{min\_of\_}RN_j$ , or

ASK USER FOR **Schema revision**: Decrease  $R.\text{min\_of\_}RN_j$

then  $R.\text{delete}(\vec{r})$ ;

If  $R$  is non-binary:  $R.\text{make\_deleted\_relationship\_consistent}(\vec{r})$ ;

else **Reject**;

$R.\text{make\_deleted\_relationship\_consistent}(\vec{r}) =$

For  $i = 1, \dots, n$ :

If  $0 < \text{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)) < R.\text{min\_of\_}RN_i$  then:

ASK USER TO CHOOSE ONE POLICY OUT OF **Reject**, **Nullify**, **Connect**, **Disconnect**,  
or **Schema revision**:

– **Reject**: Break the update.

– **Nullify**:  $\text{null}_e = (R.E\_of\_RN_i).\text{insert}(A : \text{null})$ ;

$R.\text{connect}(\vec{r}_{RN_i=\text{null}_e})$ ;

– **Connect**: ASK USER FOR A LEGAL KEY VALUE  $A : v$  for  $R.E\_of\_RN_i$ .

$e = (R.E\_of\_RN_i).\text{insert}(A : v)$ ;

$R.\text{connect}(\vec{r}_{RN_i=e})$ ; If failed, repeat **Connect**.

If  $e$  is new:  $(R.E\_of\_RN_i).\text{make\_consistent}(e)$

– **Disconnect**:

While there exists  $\vec{r}' \in \sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$  do:

$R.\text{disconnect}(\vec{r}')$

– **Schema revision**: Decrease  $R.\text{min\_of\_}RN_i$ .

## 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** is 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 is deleted from the **study** relationship type. Since **study** is non-binary, **study.make\_deleted\_relationship\_consistent**(**study.physician** :  $p$ , **study.profession** :  $f$ , **study.school** :  $s$ ) is applied. Assume, for example, that prior to the disconnection, there were exactly two **study** relationships with the physician  $p$  and the school  $s$ . Then  $min\_inconsistency_{study}(\mathbf{study.physician} : p, \mathbf{study.profession} : f, \mathbf{study.school} : s)(2) = -1$ . Then the user is asked to choose among **Reject**, **Nullify**, **Connect**, **Disconnect** or **Schema revision**. If **Nullify** is chosen, then a new null entity is inserted into 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 is 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 **study.min\_of\_study\_profession** is decreased (from 2 to 1).  $\square$

The **disconnect** method does not involve deletion of entities. Hence, since the **connect** method does not increase referential inconsistency of entities, this property is preserved. Since binary relationships are consistent by definition, their disconnection involves no propagation. That is, for binary relationship types, disconnection does not trigger neither additional disconnections, nor new insertions of entities or of relationships. Hence, the consistency of the entities in the disconnected relationship cannot decrease as this point is explicitly checked, and the consistency of existing entities cannot reduce since there are no further connections or disconnections of binary relationships. The correctness of **disconnect** is summarized in the following claim:

**Claim 3.10** *For a strongly satisfiable EER schema the **disconnect** method is correct. That is:*

1. The **disconnect** method is cardinality faithful. In particular, if  $R.\mathbf{disconnect}(\vec{r})$  terminates without rejection, the relationship  $\vec{r}$  is deleted from  $R^{\mathcal{D}}$ . Moreover,
  - (a) **disconnect** involves no entity deletions.
  - (b) If  $R$  is binary, **disconnect** does not lead to any update propagation. Its application does not invoke neither **integrity\_insert** (and its auxiliary  $E.\mathbf{make\_consistent}$  method) nor **connect** or **disconnect**.
  - (c) If  $R$  is binary, then for every component  $e_i$  of the relationship  $\vec{r}$ :  
 $\min\_inconsistency_{E_i}(e_i)([R, RN_i]) = 0$ , and  $\max\_inconsistency_{E_i}(e_i)([R, RN_i])$  stays intact (where  $E_i = R.E\_of\_RN_i$ ).
2. The **disconnect** method is terminating. Moreover, **disconnect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** In Appendix A  $\square$

### III. The reconnect method

This operation stands for a **disconnect** operation that is followed by a **connect** operation. However, under the cardinality faithfulness 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  $\{\mathbf{works\_for\_employee} : \text{Fred}, \mathbf{works\_for\_employee} : \text{Eye-department}\}$  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  $\mathcal{D}$ , 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  $\mathcal{D}$ . For simplicity, we denote the relationship  $\vec{r} = \{ RN : e, RN_1 : e_1, \dots, RN_k : e_k \}$ , where  $RN$  is the role name whose entity  $e$  should be replaced by the new entity that is identified by  $A : v$ .

**Reconnect** involves the following operations: First,  $\vec{r}$  is deleted from  $R^D$  (if it exists there). If  $R$  is binary and the deletion reduces the minimum consistency of the replaced entity  $e$ , then the user is asked to choose among several actions that either delete  $e$  or increase its minimum consistency. Then, the entity  $\bar{e}$ , identified by the key value  $A : v$  of the entity type symbol  $E = R.E\_of\_RN$  is inserted into  $E$  (if it is not already there), the new relationship  $\vec{r}/_{RN=\bar{e}}$ , which results from  $\vec{r}$  by replacing  $\bar{e}$  for the  $RN$  role, is connected, and if  $\bar{e}$  is new, it is made consistent. Finally, if  $R$  is non-binary, the effect of deletion of  $\vec{r}$  is propagated to the  $R^D$  relationship type, as in the **disconnect** method.

**Important notes:**  $e$  is not made consistent, only compensated for the lost connection. In that sense **reconnect** is consistency sensitive. The compensation for the possible decrease in minimal consistency of  $e$  is handled immediately, so that further deletions that it might invoke would not violate the insertion of the new relationship to  $R$ . On the other hand, the compensation for the deletion of  $\vec{r}$  is left to the end since the connection of  $\vec{r}/_{RN=\bar{e}}$  removes some of the possible increase in inconsistency of  $R^D$ 's relationships. In the extreme case, where  $\vec{r} = \vec{r}/_{RN=\bar{e}}$ ,  $\vec{r}/_{RN=\bar{e}}$  can be disconnected during that process. In that case the whole update is rejected.

$R.reconnect(\vec{r}, RN, A : v) =$

$R.delete(\vec{r});$

If  $R$  is binary,  $\vec{r}.RN = e$ , and  $min\_inconsistency_{R.E\_of\_RN}(e)([R, RN]) > 0$ :

ASK USER TO CHOOSE ONE POLICY OUT OF **Reject**, **Nullify**, **Connect** another relationship to  $e$ , or **Delete**  $e$ .

Assume:  $R.role\_names = \{RN, RN'\}$ ,  $E = R.E\_of\_RN$ , and  $E' = R.E\_of\_RN'$ .

– **Reject:** Break the update.

– **Nullify:**  $null\_e' = E'.insert(A' : null);$  where  $A'$  is a key attribute of  $E'$ .

$R.connect(\{RN : e, RN' : null\_e'\});$

– **Connect:** ASK USER FOR A LEGAL KEY VALUE  $A' : v'$  for  $E'$ .

$e' = E'.insert(A' : v');$

$R.connect(\{RN : e, RN' : e'\});$  If failed, repeat **Connect**.

If  $e'$  is new:  $E'.make\_consistent(e')$ .

– **Delete:**  $E.integrity\_delete(e);$

$\bar{e} = E.insert(A : v);$

$R.connect(\vec{r}/_{RN=\bar{e}});$  If the **connect** is rejected then the **reconnect** is rejected as well.

If  $\bar{e}$  is new:  $E.\mathbf{make\_consistent}(\bar{e})$ ;  
 If  $R$  is non-binary:  $R.\mathbf{make\_deleted\_relationship\_consistent}(\vec{r})$ ;  
 If  $\mathbf{not}(R.\mathbf{retrieve}(\vec{r}_{RN=\bar{e}}))$ : Break the update.

### Example

Consider the EER schema from Figure 1. In order to reconnect the **treating** relationship  $\{\mathbf{treating.physician} : p, \mathbf{treating.visit} : v\}$ , to a new physician, identified by the key value  $license\_no : ph5678$  of **Physician**,  $\mathbf{treating.reconnect}(\{\mathbf{treating.physician} : p, \mathbf{treating.visit} : v\}, \mathbf{treating.physician}, license\_no : ph5678)$  is applied. If  $v$  is the only visit of the physician entity  $p$ , then after the deletion of  $\{\mathbf{treating.physician} : p, \mathbf{treating.visit} : v\}$   $p$  becomes minimally inconsistent as it must have at least one visit. Then the user is 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  $ph5678$  is inserted into **Physician** (if it is not already there), the new **treating** relationship  $\{\mathbf{treating.physician} : p', \mathbf{treating.visit} : v\}$  is connected, and  $p'$  is made consistent if it is new.

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

The correctness of **reconnect** is summarized in the following claim:

**Claim 3.11** *For a strongly satisfiable EER schema the **reconnect** method is correct. That is:*

1. *The **reconnect** method is cardinality faithful. In particular, if*

$R.\mathbf{reconnect}(\vec{r}, RN, A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $R.E\_of\_RN$  is inserted into  $(R.E\_of\_RN)^{\mathcal{D}}$  (if it is not already there), the relationship  $\vec{r}$  is deleted from  $R^{\mathcal{D}}$ , and the relationship  $\vec{r}_{RN=\bar{e}}$  is inserted into  $R^{\mathcal{D}}$ .

2. The **reconnect** method is terminating. Moreover, **reconnect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** In Appendix A  $\square$

The cardinality faithfulness property of the integrity methods introduced in this paper is summarized as follows:

**Property:** If  $\mathcal{D}$  is a proper data base instance of a strongly satisfiable EER schema  $\mathcal{ER}$ , and  $\mathcal{D}'$  results from  $\mathcal{D}$  by the application of an integrity method, then  $\mathcal{D}'$  is *consistent up to null entities*. That is,  $\mathcal{D}'$  satisfies all key, relationship construct and sub-typing constraints, and all cardinality constraints set on real entities. If  $\mathcal{D}'$  includes no null entities, then it is a consistent database instance of  $\mathcal{ER}$ .  $\square$

## 4 Conclusion

In this paper we introduced an extension of the EER data model with integrity sensitive update methods. For that purpose we first formalized the EER model, and clarified the semantics of its cardinality constraints. The structure methods are proved to be correct. That is, they are terminating, cardinality faithful, and carefully designed to be cardinality sensitive, so not to invoke unnecessary propagation. The definition of the structure methods is independent of any implementation of the EER schema, since they are fully defined by the cardinality constraints, using the primitive update methods. That is, the primitive methods add an extra layer of abstraction, such that a translation of MEER into any implementation schema can be defined in terms of the primitive methods alone. The integrity methods should stay intact. The contribution of this research is in capturing the intended meaning of the cardinality constraints as an integral part of the EER schema.

Consequently, the structure methods can be integrated into any system that uses the EER schema on its conceptual level. For relational or object-oriented databases, the conventional mappings of an EER schema into a relational or an object-oriented schema can be extended with

automatic creation of the structure methods. The extension is obtained through mappings of the primitive methods. The mapping of the more complicated structure methods should be obtained directly from the primitive methods mapping<sup>3</sup>.

Similarly, the structure methods can be integrated into semi-structured data systems that use an EER schema to direct the construction of their semi-structured schema (see, for example, [20]). Another venue is CASE tools like Rose [2] and Rhapsody [15], that synthesize code or prototype files, based on visual diagrams that include EER [23]. Clearly, the structure methods can be rendered into code in a straightforward manner.

In the future we plan to extend the structure methods to handle more flexible kinds of cardinality constraints (e.g., [16]), and other constraints as well. For that purpose, we plan to formalize the repair language of MEER so that it lends itself to further formal extension and analysis.

## References

- [1] M. Balaban and P. Shoval. Resolving the "weak status" of weak entity types in entity relationship schemas. In *Entity-Relationship Approach (ER'99)*, pages 367–383. North-Holland, 1999.
- [2] G. Booch, I. Jacobson, and J. Rumbaugh. *The UML Specification Documents*. Rational Software Corp., 1997.
- [3] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [4] M. Bouzeghoub and E. Metais. Semantic modeling and object modeling: Two complementary paradigms. In *Entity-Relationship Approach (ER'91)*, pages 325–348. North-Holland, 1991.
- [5] M. Bouzeghoub and E. Metais. Semantic modeling of object oriented databases. In *Very Large Data Bases (VLDB'91)*, pages 3–14, September 1991.
- [6] D. Calvanese, M. Lenzerini, and D. Nardi. Description logics for conceptual data modeling. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*. Kluwer Academic Publishers, 1998.

---

<sup>3</sup>We are currently in the process of extending the conventional EER-to-OO schema mappings to map the newly added methods. The method mapping will be defined in terms of the primitive methods alone.

- [7] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *VLDB'90*, pages 566 – 577, 1990.
- [8] P.P. Chen. The entity-relationship model: toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.
- [9] K.R. Dittrich. Object-oriented database systems. In S. Spaccapietra, editor, *Entity-Relationship Approach*, pages 51–66. Elsevier Science Publishers, 1987.
- [10] R. Elmasri and S.B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Company, 1994.
- [11] C. Fahrner and G. Vossen. A survey of database design transformations based on the entity-relationship model. *Data and Knowledge Engineering*, pages 213 – 250, 1995.
- [12] S. Ferg. Cardinality concepts in entity-relationship modeling. In E. Teorey, editor, *Entity-Relationship Approach*. North-Holland, 1991.
- [13] T. Feyer and B. Thalheim. E/R based scenario modeling for rapid prototyping of web information services. In *ER'99 Workshop on the World Wide Web and Conceptual Modeling*, pages 353–263, november 1999.
- [14] J. Gomez, C. Cachero, and D. Pastor. Extending a conceptual modeling approach to web application design. In *CAiSE-2000*, pages 79–93. North-Holland, 2000.
- [15] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 1997.
- [16] S. Hartmann. On the consistency of int-cardinality constraints. In *Entity-Relationship Approach (ER'98)*. North-Holland, 1998.
- [17] Y. Kornatzky and P. Shoval. Conceptual design of object-oriented database schemas using the binary-relationship model. *Data and Knowledge Engineering*, 14:265–288, 1994.
- [18] B. Lazarevic and V. Misic. Extending the entity-relationship model to capture dynamic behaviour. *European Journal of Information Systems*, 1(2):95–106, 1991.
- [19] C. Lecluse, P. Richard, and F. Velez. An object-oriented data model. Technical Report 10, Altair, 1987.

- [20] S.Y. Lee, M.L. Lee, T.W. Ling, and L.A. Kalinichenko. Designing good semi-structured databases. In *Entity-Relationship Approach (ER'99)*, pages 131–145. North-Holland, 1999.
- [21] M. Lenzerini and P. Nobili. On the satisfiability of dependency constraints in entity-relationship schemata. *Information Systems*, 15(4):453–461, 1990.
- [22] M. Lenzerini and G. Santucci. Cardinality constraints in the entity-relationship model. In Davis, Jejo dia, Ng, and Yeh, editors, *Entity-Relationship Approach*. North-Holland, 1983.
- [23] P. Loucopoulos and R. Zicari. *Conceptual Modeling, Databases, and CASE*. Wiley, 1992.
- [24] D.E. Monarchi and J.R. Smith. The representation of rules in the ER model. *Data and Knowledge Engineering*, pages 45 – 61, 1992/93.
- [25] P. Scheuermann, G. Schiffner, and H. Weber. Abstraction capabilities and invariant properties modeling within the entity- relationship approach. In *Entity-Relationship Approach*, pages 121–140. North-Holland, 1979.
- [26] A.K. Tanaka, S.B. Navathe, S. Chakravarthy, and K. Karlapalem. ER-R: An enhanced ER model with situation-action rules to capture application semantics. In *Entity-Relationship Approach (ER'91)*, pages 59–75. North-Holland, 1991.
- [27] B. Thalheim. Fundamentals of cardinality constraints. In *Entity-Relationship Approach*, pages 7–23. North-Holland, 1992.
- [28] B. Thalheim. *Fundamentals of Entity-Relationship Modeling*. Springer-Verlag, 1998.
- [29] J. Widom and S. Ceri. *Active Database Systems*. Morgan-Kaufmann, 1996.

## A Appendix: Proofs

### 1 Correctness of the `integrity_insert` method

**Claim 3.7** For a strongly satisfiable EER schema the `integrity_insert` method is correct. That is:

1. The `integrity_insert` method is cardinality faithful. In particular, if  $E.integrity\_insert(A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $E$  is inserted to  $E^{\mathcal{D}}$ .
2. The `integrity_insert` method is terminating. Moreover, `integrity_insert` is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** For the purpose of this proof we first manually expand all nested calls to **connect** by the body of **connect**, with arguments appropriately substituted. Otherwise, we would have to carry out a simultaneous inductive proof for both the **integrity\_insert** and the **connect** methods. Since **connect** is called only with binary relationships which are consistent by definition, we omit the calls to  $R.\mathbf{make\_consistent}$  from the expanded body of **connect**, as these calls succeed immediately. So, the call  $R.\mathbf{connect}(\{RN : e, RN' : null\_e\})$  is expanded into:

If  $E.\mathbf{retrieve}(e)$  and  $E'.\mathbf{retrieve}(null\_e)$

and  $\mathbf{cardinality}(\sigma_{RN=e}(R)) < R.\mathbf{max\_of\_RN'}$  and  $\mathbf{cardinality}(\sigma_{RN'=null\_e}(R)) < R.\mathbf{max\_of\_RN}$

then  $R.\mathbf{insert}(\{RN : e, RN' : null\_e\})$ ;

and the call  $R.\mathbf{connect}(\{RN : e, RN' : e'\})$  is expanded into:

If  $E.\mathbf{retrieve}(e)$  and  $E'.\mathbf{retrieve}(e')$

and  $\mathbf{cardinality}(\sigma_{RN=e}(R)) < R.\mathbf{max\_of\_RN'}$  and  $\mathbf{cardinality}(\sigma_{RN'=e'}(R)) < R.\mathbf{max\_of\_RN}$

then  $R.\mathbf{insert}(\{RN : e, RN' : e'\})$ ;

As a result, **integrity\_insert** involves only calls to the primitive methods  $E.\mathbf{retrieve}$ ,  $E.\mathbf{insert}$ , and  $R.\mathbf{insert}$  for binary relationships, and to the auxiliary update method  $E.\mathbf{make\_consistent}$ .

The **integrity\_insert** method does not involve explicit deletions of entities or of relationships. Hence we need to prove only the items concerning new insertions and existing instances.

1. **Relationships** – Only binary relationships, which are consistent by definition, are inserted.
2. **Existing entities** – Every insertion of a binary relationship is preceded by a test that it will not violate maximum cardinality constraints set on the related entities. In case of violation the insertion does not take place. Hence, the consistency of existing entities can only increase, since their *max\_inconsistencies* stay intact, while their *min\_inconsistencies* can decrease.
3. **New entities** – Their consistency follows immediately from Claim A.1 below.

Termination of **integrity\_insert** depends on the termination of  $E.\mathbf{make\_consistent}$ . This is proved in Claim A.2 below (starting from a consistent database instance, the resulting instance after a single insertion satisfies the requirements in the Claim).  $\square$

**Claim A.1** *When  $E.\mathbf{make\_consistent}(e)$  terminates without rejection,  $\mathbf{min\_inconsistency}_E(e) = 0$ ,  $\mathbf{max\_inconsistency}_E(e)$  stays intact, all newly inserted entities are consistent, and existing entities and relationships can become only more consistent.*

**Proof:** The proof is carried over the expanded method as above, so that there are no nested calls to **connect**. The proof is by induction on the depth  $n$  of nested calls to entity **make\_consistent**.

$n = 0$  In every while round either  $e$  is connected to a new null entity, or it is connected to an existing entity, or the schema is revised. In the first case,  $\text{min\_inconsistency}_E(e)$  decreases, and the new null entity is consistent, by definition. In the second case,  $\text{min\_inconsistency}$  of both entities decreases. In the third case,  $\text{min\_inconsistency}_E(e)$  decreases due to the schema revision. Consequently, the while loop terminates, and upon termination  $\text{min\_inconsistency}_E(e) = 0$ .

$n > 0$  Since  $n > 0$  there is at least one while round in which **Propagate** is selected, and a new entity is inserted. In that round, if it succeeds,  $e$  is connected to a new entity  $e'$  of entity type  $E'$ . After the new binary relationship is inserted,  $\text{min\_inconsistency}_E(e)$  decreases,  $\text{max\_inconsistency}_E(e)$  stays intact, and  $e'$  can become only more consistent. By the inductive hypothesis, when  $E'.\text{make\_consistent}(e')$  terminates,  $\text{min\_inconsistency}_{E'}(e') = 0$ ,  $\text{max\_inconsistency}_{E'}(e')$  stays intact, all newly inserted entities are consistent, and existing entities and relationships can become only more consistent. Since  $e'$  is a new entity,  $\text{max\_inconsistency}_{E'}(e') = 0$  and hence  $e'$  is consistent.

In sum, when a while round terminates,  $\text{min\_inconsistency}_E(e)$  decreases,  $\text{max\_inconsistency}_E(e)$  stays intact, all newly inserted entities are consistent, and existing entities and relationships can become only more consistent. Since the while terminates when  $\text{min\_inconsistency}_E(e) = 0$ , the claim holds.

□

**Claim A.2** *Let  $\mathcal{E}$  be a strongly satisfiable EER schema,  $\mathcal{D}$  be a database instance that does not violate maximum cardinality constraints, and  $E$  be an entity type symbol in  $\mathcal{E}$ . Then,  $E.\text{make\_consistent}$  is terminating with respect to  $\mathcal{D}$ . That is, any application of  $E.\text{make\_consistent}$  includes terminating branches.*

**Proof:** Lenzerini and Nobili, in [21], introduce a system of linear inequalities that is built per schema, such that the schema is strongly satisfiable if and only if the system has a solution, and every consistent database instance corresponds to a solution to the system. The EER model discussed in [21] includes only participation cardinality constraints, and allows no subtyping. Since  $E.\text{make\_consistent}$  methods ignore non-binary relationship type symbols, they can be removed from the schema without affecting the method operation. For the remaining binary relationship type symbols, participation and look-across cardinality constraints are the same.

Following Proposition 2.1, consider the transformed schema  $\mathcal{E}'$  that includes no sub-typing.  $\mathcal{E}'$  is strongly satisfiable since  $\mathcal{E}$  is strongly satisfiable. The system of inequalities that [21] associates with  $\mathcal{E}'$ , denoted  $\Psi^{\mathcal{E}'}$ , includes the following inequalities:

1. For every relationship construct  $R(RN_1 : E_1[\text{min}_1, \text{max}_1], RN_2 : E_2[\text{min}_2, \text{max}_2])$ , the inequalities are  $\text{max}_2 \times E_1 \geq R \geq \text{min}_2 \times E_1$  and  $\text{max}_1 \times E_2 \geq R \geq \text{min}_1 \times E_2$ .
2. For every entity or relationship type symbol  $T$ , the inequality  $T > 0$ .

For every consistent database instance of  $\mathcal{E}'$  with non-empty types, the sizes of the types form a solution to  $\Psi^{\mathcal{E}'}$  (when substituted, respectively, for the type symbols in  $\Psi^{\mathcal{E}'}$ ).

Let  $\mathcal{D}'$  be the database instance of  $\mathcal{E}'$  that is obtained from  $\mathcal{D}$ :  $\mathcal{D}'$  is the restriction of  $\mathcal{D}$  to the remaining types, and interpreting the relationship symbols that replace subtyping as the identity relationship on the extension of the subtype. Clearly,  $\mathcal{D}'$  does not violate maximum cardinality constraints in  $\mathcal{E}'$ . Consider an application of  $E$ .**make\_consistent** under a *minimal insertion* policy (a new entity is inserted only if no existing one can be used for a missing connection). It initiates a process of while rounds. When applying the **Propagate** policy, each round considers, for an entity  $e$  of an entity type  $E^{\mathcal{D}'}$ , one missing relationship through a relationship type  $R^{\mathcal{D}'}$  to an entity  $e'$  of an entity type  $E'^{\mathcal{D}'}$ . Let  $(n, m)$  be the participation cardinality constraint for  $E$  in  $R$ , and let  $|T|$  denote the current size of the type assigned to a type symbol  $T$  in the database instance. Then, since  $\mathcal{D}'$  does not violate maximum cardinality constraints, it means that  $|R| < n \times |E|$ , and if  $e'$  is necessarily a newly inserted entity of  $E'$  (*minimal insertion*), then also  $m \times |E'| < |R|$ . That is, the missing relationship is reflected in the violation of the  $\Psi^{\mathcal{E}'}$  inequalities  $m \times E' \geq R \geq n \times E$ , and the insertion of a relationship and possibly also of an entity.

Let  $\{\hat{E}_1, \dots, \hat{E}_n, \hat{R}_1, \dots, \hat{R}_m\}$  be an integer solution to  $\Psi^{\mathcal{E}'}$ , satisfying  $|E_i^{\mathcal{D}'}| \leq \hat{E}_i$  ( $1 \leq i \leq n$ ) and  $|R_j^{\mathcal{D}'}| \leq \hat{R}_j$  ( $1 \leq j \leq m$ )<sup>4</sup>. We claim that during the application of  $E$ .**make\_consistent** under the minimal insertion propagation policy, the sizes of the types are bounded from above by this solution. That is, for every type symbol  $T$ ,  $|T| \leq \hat{T}$  at any point of this process. Suppose that this claim does not hold. Then, since in every round a new relationship is inserted, let  $R$  be the relationship type symbol which is the first to satisfy  $|R| > \hat{R}$ . Then, for some inequality  $R \geq n \times E$  in  $\Psi^{\mathcal{E}'}$  we have, at that point:  $|R| - 1 < n \times |E|$  (the inequality did not hold prior to the relationship insertion). But, then we have:

$$\hat{R} \leq |R| - 1 < n \times |E| \leq n \times \hat{E}$$

which contradicts the assumption about  $\hat{R}$  and  $\hat{E}$  being part of a solution to  $\Psi^{\mathcal{E}'}$ .

Similarly, if  $E'$  is the entity type symbol which is the first to satisfy  $|E'| > \hat{E}'$ . Then, for some inequalities  $m \times E' \geq R \geq n \times E$  in  $\Psi^{\mathcal{E}'}$  we have, at that point:  $m \times |E'| - 1 < |R| - 1 < n \times |E|$  (the inequality did not hold prior to the relationship and the entity insertions). But, then we have:

$$m \times \hat{E}' \leq m \times |E'| - 1 < |R| - 1 < n \times |E| \leq n \times \hat{E}$$

which contradicts the assumption about  $\hat{E}'$  and  $\hat{E}$  being part of a solution to  $\Psi^{\mathcal{E}'}$ .

Altogether, while in every round a relationship and possibly an entity is inserted, all type sizes are bounded by the selected solution to  $\Psi^{\mathcal{E}'}$ . Therefore, the process must terminate.

□

---

<sup>4</sup>An integer solution exists since the system of inequalities is homogeneous [21].

## 2 Correctness of the `integrity_delete` method

**Claim 3.8** For a strongly satisfiable EER schema the `integrity_delete` method is correct. That is:

1. The `integrity_delete` method is cardinality faithful. In particular, if  $E.\text{integrity\_delete}(A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $E$ , if it exists, is deleted from  $E^{\mathcal{D}}$ .
2. The `integrity_delete` method is terminating. Moreover, `integrity_delete` is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** The `integrity_delete` method does not involve explicit insertions of entities or of relationships, and there are no explicit deletions of non-binary relationships. These updates are done only through the integrity methods `reconnect` and `disconnect` – for non-binary relationships. Since both methods are cardinality faithful (Claims 3.10 and 3.11), newly inserted instances are consistent, referential inconsistencies of existing entities can only decrease, and the applications of these methods do not cause new inconsistencies, and can only decrease existing inconsistencies. Moreover, since `reconnect` and `disconnect` are also terminating, so is `integrity_delete`.

Hence we need to prove only the items concerning deleted entities and existing instances, i.e., that deleted entities are referentially consistent upon termination, and existing entities and relationships can become only more consistent. These items follow immediately from the claim about the auxiliary method `E.make_referentially_consistent`, below.  $\square$

**Claim A.3** *When `E.make_referentially_consistent(e)` terminates without rejection,  $e$  is referentially consistent with respect to  $E$ , all entities deleted during its operation are referentially consistent with respect to the entity types from which they are deleted, and existing entities and relationships can become only more consistent.*

**Proof:** Induction on the depth  $n$  of nested calls to `E.make_referentially_consistent`. Since no integrity method, beyond `integrity_delete` invokes `make_referentially_consistent`, the only call to `make_referentially_consistent` is the one made by the nested invocation of `integrity_delete`.

$n = 0$  In every while round a relationship  $\vec{r} \in e.\text{rels}([R, RN])$  that includes  $e$  as its  $RN$  component is either reconnected or disconnected. The reconnection may be either to a new null entity in  $E^{\mathcal{D}}$  or to a real, newly inserted or existing, entity in  $E^{\mathcal{D}}$ . A non-binary relationship can be disconnected unconditionally, but a binary relationship can be deleted only provided that it does not cause violation of minimum restrictions, or the schema is revised. Otherwise, there is a nested call to `make_referentially_consistent`. Since both `reconnect` and `disconnect` are integrity faithful methods (that, anyhow, do not involve entity deletions), they do not create new inconsistencies, and can only decrease existing ones. A binary relationship is directly deleted (without recursive invocation

of **integrity\_delete**), only in case that the related entities stay minimally consistent. In every while round at least one reference to  $e$  is removed since:

- At least one relationship  $\vec{r}$  that involves  $e$  is deleted.
- Newly connected relationships (through **disconnect** or **reconnect**) can involve either existing entities or newly inserted ones.
- The entity  $e$  is already deleted and the primitive method **insert** for entities always creates new entities<sup>5</sup>.

Consequently, the while loop terminates, and upon termination

$$referential\_inconsistency_E(e) = 0.$$

$n > 0$  Since  $n > 0$  there is at least one while round in which **Delete** is selected, and applied to a binary relationship, thereby leading to or increasing violation of minimum cardinality constraints set on the related entities. In that round,  $\vec{r}$  is deleted, and **integrity\_delete** is recursively called on entity type  $E'$  and entity  $e'$ . Explicit expansion of **integrity\_delete** yields direct deletion of  $e'$  and the call  $E'.\mathbf{make\_referentially\_consistent}(e')$ . By the inductive hypothesis, when  $E'.\mathbf{make\_referentially\_consistent}(e')$  terminates without rejection,  $e'$  is referentially consistent with respect to  $E'$ , all entities deleted during its operation are referentially consistent with respect to the entity types from which they are deleted, and existing entities and relationships can become only more consistent. Hence, in that while round, at least one reference to  $e$  is removed due to the deletion of  $\vec{r}$ , and the rest of the claim still holds. In sum, for any while round, if it terminates without rejection,  $referential\_inconsistency_E(e)$  decreases, and the rest of the claim holds. Since the while terminates when  $referential\_inconsistency_E(e) = 0$ , the claim is proved.

□

### 3 Correctness of the connect method

**Claim 3.9** For a strongly satisfiable EER schema the **connect** method is correct. That is:

1. The **connect** method is cardinality faithful. In particular, if  $R.\mathbf{connect}(\vec{r})$  terminates without rejection, the relationship  $\vec{r}$  is inserted into  $R^D$ . Moreover,
  - (a) **connect** propagates no deletions of either entities or relationships. Consequently, for every component  $e_i$  of the relationship  $\vec{r}$ ,  $e_i.no\_of\_rels([R, RN_i])$  increases. If  $R$  is binary, then  $min\_inconsistency_{R.E\_of\_RN_i}(e_i)$  decreases while  $max\_inconsistency_{R.E\_of\_RN_i}(e_i)$  stays intact.

---

<sup>5</sup>Note that in theory, a user can provide for re-connections the same key value as that of the deleted entity. This is not a rational user behavior, but is still possible. In any case, a new entity identified by the same key value will be created.

(b) If  $R$  is binary, **connect** does not lead to any update propagation. Its application does not invoke neither **integrity\_insert** (and its auxiliary  $E$ .**make\_consistent** method) nor **connect** itself.

2. The **connect** method is terminating. Moreover, **connect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** Termination of **connect** is proved in Claim A.5 below. For the purpose of proving cardinality faithfulness we first manually expand all nested calls to **connect** by the body of **connect**, with arguments appropriately substituted. As a result, **connect** involves only calls to the primitive methods  $E$ .**retrieve**,  $E$ .**insert**, and  $R$ .**insert**, and to the auxiliary update methods  $E$ .**make\_consistent** and  $R$ .**make\_consistent**.

The **connect** and  $R$ .**make\_consistent** methods do not involve explicit deletions of entities or of relationships, and all relationship insertions are preceded with tests that the related entities do exist. This claim holds also for the auxiliary update method  $E$ .**make\_consistent**. Hence, **connect** cannot create new referential inconsistencies, and existing ones stay intact. We need to prove only the cardinality faithfulness items concerning new insertions and existing instances.

1.  $R$  is binary – Since binary relationships are consistent, by definition,

$\min\_inconsistency_R(\vec{r})(i) = 0$  for  $i = 1, 2$ , and  $R$ .**make\_consistent**( $\vec{r}$ ) succeeds immediately. Hence, the **connect** involves only the tests that the related entities exist, and that the new relationship can be connected without violating maximum cardinality constraints set on the related entities. If the tests succeed, then  $\vec{r}$  is inserted into  $R^D$ . There are no newly inserted entities, the inserted relationship is consistent, and the minimal inconsistency of related entities can only increase (in case that they were minimal inconsistent before), while their maximal inconsistency stay intact.

2.  $R$  is non-binary – The **connect** tests that the related entities do exist, and the requested relationship insertion would not violate maximum cardinality restrictions set on relationships in  $R$ . In that case, the relationship  $\vec{r}$  is inserted, and  $R$ .**make\_consistent**( $\vec{r}$ ) is invoked. Since all explicit relationship insertions within **connect** and  $R$ .**make\_consistent** are for non-binary relationships, they do not affect the consistency of the related entities (consistency of entities depend only on their binary relationships). By the claim of the auxiliary update method  $E$ .**make\_consistent** that is invoked (Claim A.1) its application can only increase the consistency of existing entities and relationships. Hence it is left to prove that newly inserted entities and relationships are consistent, and that  $\vec{r}$  is minimally consistent. This follows from the claim about the auxiliary method  $R$ .**make\_consistent**, below (Claim A.4).

The additional items in the claim follow immediately from the above.  $\square$

**Claim A.4** When  $R$ .**make\_consistent**( $\vec{r}$ ) terminates without rejection,

$min\_inconsistency_R(\vec{r}) = 0$ ,  $max\_inconsistency_R(\vec{r})$  stays intact, and all newly inserted entities and relationships are consistent.

**Proof:** The proof is carried over the expanded version of  $R.\mathbf{make\_consistent}$ , so that there are no nested calls to **connect**. The proof is by induction on the depth  $n$  of nested calls to  $R.\mathbf{make\_consistent}$ .

$n = 0$  In every while round the schema is revised, such that a minimum cardinality constraint is relaxed.

Hence in every while round  $min\_inconsistency_R(\vec{r})$  increases and there are no insertions. The while terminates when  $min\_inconsistency_R(\vec{r}) = 0$ . So the claim holds.

$n > 0$  Since  $n > 0$  there is at least one while round in which either **Nullify** or **Connect** is selected, and a new relationship is inserted.

1. If **Nullify** is selected, then a new null entity is inserted, a new relationship that does not violate maximum cardinality constraints on  $R$  is inserted, and made consistent by applying  $R.\mathbf{make\_consistent}$  to it. The new relationship causes  $min\_inconsistency_R(\vec{r})$  to increase, and by the inductive hypothesis the new relationship, and all newly inserted entities and relationships are consistent. The new null entity is consistent by definition.
2. If **Connect** is selected, then a new relationship that does not violate maximum cardinality constraints on  $R$  is inserted, and made consistent by applying  $R.\mathbf{make\_consistent}$  to it. By the inductive hypothesis, the new relationship and all newly inserted entities and relationships are consistent. The new relationship involves either an existing entity suggested by the user, or a newly inserted entity  $e$ . In the latter case,  $e$  is made consistent by applying  $E.\mathbf{make\_consistent}$  to it. By Claim A.1,  $e$  is consistent, all newly inserted entities and relationships are consistent, and existing inconsistencies can only decrease. In either case, the insertion of the new relationship to  $R$  causes  $min\_inconsistency_R(\vec{r})$  to increase.

Hence, in every while round  $min\_inconsistency_R(\vec{r})$  increases, and all newly inserted entities and relationships are consistent. The while terminates when  $min\_inconsistency_R(\vec{r}) = 0$ . So the claim holds.

□

**Claim A.5** Let  $\mathcal{E}$  be a strongly satisfiable EER schema,  $\mathcal{D}$  be a database instance that does not violate maximum cardinality constraints, and  $R$  be a relationship type symbol in  $\mathcal{E}$ . Then,  $R.\mathbf{connect}$  is terminating with respect to  $\mathcal{D}$ .

**Proof:** If  $R$  is binary, then the newly inserted relationship  $\vec{r}$  is consistent and the application terminates. For a non-binary relationship type symbol  $R$  we simulate  $R.\mathbf{connect}$  by a sequence of terminating operations

in a schema with binary relationship types alone. We use the transformation  $\tau$  introduced in Definition 2.2 in Subsection 2.3, that replaces a non-binary relationship type symbol by a set of new entity type symbols and new binary relationship type symbols.

The operation  $R.\mathbf{connect}(\vec{r})$  consists of two operations:  $R.\mathbf{insert}(\vec{r})$  and  $R.\mathbf{make\_consistent}(\vec{r})$ . In  $\tau(\mathcal{E})$ ,  $R.\mathbf{insert}(\vec{r})$  is simulated by operations that insert all the  $n-1$  projections of  $\vec{r}$  to the new entity types  $ER_1, \dots, ER_n$ , and connect these tuples to the entity components of  $\vec{r}$ . Let  $\vec{r} = \{RN_1 : e_1, \dots, RN_n : e_n\}$ , and denote  $\vec{r} - e_i$  by  $\vec{r}_i$ . Then,  $R.\mathbf{insert}(\vec{r})$  is simulated by:  $ER_1.\mathbf{insert}(r_1), \dots, ER_n.\mathbf{insert}(r_n)$  and  $R_{i,j}.\mathbf{connect}(r_i, e_j)$ ,  $1 \leq i, j \leq n$ . The operation  $R.\mathbf{make\_consistent}(\vec{r})$  is simulated by the operations  $ER_i.\mathbf{make\_consistent}(\vec{r}_i)$ ,  $1 \leq i \leq n$ .

By Proposition 2.3,  $\tau(\mathcal{E})$  is strongly satisfiable since  $\mathcal{E}$  is so. Hence, the applications of  $ER_i.\mathbf{make\_consistent}$  are terminating. Since there is a direct correspondence between the steps of  $R.\mathbf{make\_consistent}(\vec{r})$  to the steps of the simulating operations, and the latter are terminating, the simulated operation  $R.\mathbf{connect}(\vec{r})$  is also terminating.  $\square$

#### 4 Correctness of the disconnect method

**Claim 3.10** For a strongly satisfiable EER schema the **disconnect** method is correct. That is:

1. The **disconnect** method is cardinality faithful. In particular, if  $R.\mathbf{disconnect}(\vec{r})$  terminates without rejection, the relationship  $\vec{r}$  is deleted from  $R^{\mathcal{D}}$ . Moreover,

- (a) **disconnect** involves no entity deletions.
- (b) If  $R$  is binary, **disconnect** does not lead to any update propagation. Its application does not invoke neither **integrity\_insert** (and its auxiliary  $E.\mathbf{make\_consistent}$  method) nor **connect** or **disconnect**.
- (c) If  $R$  is binary, then for every component  $e_i$  of the relationship  $\vec{r}$ :  
 $\min\_inconsistency_{E_i}(e_i)([R, RN_i]) = 0$ , and  $\max\_inconsistency_{E_i}(e_i)([R, RN_i])$  stays intact (where  $E_i = R.E\_of\_RN_i$ ).

2. The **disconnect** method is terminating. Moreover, **disconnect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** Termination is proved in a similar way to the proof of termination for the **connect** method. For a binary relationship type the method terminates immediately. For a non-binary one, the method is simulated by terminating methods over the transformed schema.

For the purpose of this proof we first manually expand the nested call to **disconnect** by the body of **disconnect**, with arguments appropriately substituted. As a result, **disconnect** involves only calls to the primitive methods  $E.\mathbf{insert}$ , and  $R.\mathbf{delete}$ , to the integrity method **connect**, and to the auxiliary update methods  $E.\mathbf{make\_consistent}$  and  $R.\mathbf{make\_deleted\_relationship\_consistent}$ .

The **disconnect** and **R.make\_deleted\_relationship\_consistent** methods do not involve explicit deletions of entities, and the **connect** and **E.make\_consistent** methods were proved not to create new referential inconsistencies, and leave existing ones intact. Hence, the **disconnect** method can only decrease existing referential inconsistencies through disconnection of relationships, and creates no new ones. We need to prove only the cardinality faithfulness items concerning new insertions and existing instances.

1. **R is binary – disconnect** involves only the test that the relationship can be disconnected without violating minimum cardinality constraints set on the related entities. If the test succeeds, or the user enforces its success by schema revision, then  $\vec{r}$  is deleted from  $R^D$ . There are no newly inserted entities or relationships, and since  $\vec{r}$  passed the minimum cardinality test, then for every component  $e_i$  of  $\vec{r}$ ,  $\min\_inconsistency_{R.E\_of\_RN_i}(e_i)([R, RN_i]) = 0$ , while  $\max\_inconsistency_{R.E\_of\_RN_i}(e_i)$  stays intact. Since  $R$  is binary, all of its relationships are consistent, and the removal of  $\vec{r}$  does not create new inconsistencies.
2. **R is non-binary –** The relationship  $\vec{r}$  is deleted, and **R.make\_deleted\_relationship\_consistent** is invoked. Since explicit relationship deletions within **disconnect** and **make\_deleted\_relationship\_consistent** are for non-binary relationships, they do not affect the consistency of the related entities (consistency of entities depend only on their binary relationships). By the claims of **E.make\_consistent** and of **connect** (Claims A.1 and 3.9), their applications can only increase the consistency of existing entities and relationships. Hence it is left to prove that any increase in relationship inconsistency that is caused by the deletion of  $\vec{r}$  is compensated, and during that process, newly inserted entities and relationships are made consistent. This is proved in the claim about the auxiliary method **R.make\_deleted\_relationship\_consistent**, below.

The additional items in the claim follow immediately from the above.  $\square$

**Claim A.6** *When **R.make\_deleted\_relationship\_consistent**( $\vec{r}$ ) terminates without rejection, then:*

1. *Any increase in relationship inconsistency that is caused by the deletion of  $\vec{r}$ , directly or indirectly, is compensated. More precisely, for  $i = 1, \dots, n$ , if in the beginning, for some  $\vec{r}' \in \sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$ ,  $\min\_inconsistency_R(\vec{r}')(i) < 0$ , then upon termination, either it is increased or  $\vec{r}'$  is deleted, and the impact of the deletion is propagated (so to compensate for possible new inconsistency increases).*
2. *All newly inserted entities and relationships are consistent.*

**Proof:** The proof is carried over the expanded version of **R.make\_deleted\_relationship\_consistent**, so that there are no nested calls to **disconnect**. The proof is

by induction on the depth  $n$  of nested calls to

***R.make\_deleted\_relationship\_consistent***.

The only relationships in  $R$  whose consistency can be directly affected by the deletion of  $\vec{r}$  are relationships in  $\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$  ( $i = 1, \dots, n$ ), since for every  $1 \leq i \leq n$ , one relationship is deleted. If following the deletion of  $\vec{r}$ ,

$0 < \text{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)) < R.\text{min\_of\_}RN_i$ , for some  $i = 1, \dots, n$ , then the deletion can be compensated by an insertion of a relationship to

$\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$ , so that the consistency of relationships in this set does not decrease. Alternatively, the whole set can be deleted, and the effect of the deletions should be propagated.

$n = 0$  For every  $i$ , if there exists a relationship  $\vec{r}'$  in

$\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$ , for which  $\text{min\_inconsistency}_R(\vec{r}')(i) < 0$ , then one of **Nullify**, **Connect** or **Schema revision** is selected by the user. If the schema is revised, then a minimum cardinality constraint is relaxed. In **Nullify** and **Connect** a new relationship, that causes  $\text{cardinality}(\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R))$  to increase, is inserted using **connect**, and newly inserted entities are made consistent using **E.make\_consistent**. Hence,  $\text{min\_inconsistency}_R(\vec{r}')(i)$  properly increases. Since both **connect** and **E.make\_consistent** are cardinality faithful, newly inserted entities and relationships are consistent, and existing inconsistencies can only decrease. Since there are no deletions of relationships, no other relationship inconsistencies are created or increased. Hence the claim holds.

$n > 0$  Since  $n > 0$  there is at least one  $i$  for which **Disconnect** is selected, and all relationships in

$\sigma_{RN_1=e_1, \dots, RN_{i-1}=e_{i-1}, RN_{i+1}=e_{i+1}, \dots, RN_n=e_n}(R)$  are deleted from  $R$ , and the impacts of these deletions are propagated using **R.make\_deleted\_relationship\_consistent**. So, for this  $i$ , all relationships whose inconsistency increases due to the deletion of  $\vec{r}$ , are deleted. By the inductive hypothesis, the application of

**R.make\_deleted\_relationship\_consistent** to each of the deleted relationships removes all indirect increase in relationship inconsistencies, and all newly inserted entities and relationships are consistent. So the claim holds.

□

## 5 Correctness of the reconnect method

**Claim 3.11** For a strongly satisfiable EER schema the **reconnect** method is correct. That is:

1. The **reconnect** method is cardinality faithful. In particular, if

$R.\text{reconnect}(\vec{r}, RN, A : v)$  terminates without rejection, the entity identified by the key value  $A : v$  of  $R.E\_of\_RN$  is inserted into  $(R.E\_of\_RN)^{\mathcal{D}}$  (if it is not already there), the relationship  $\vec{r}$  is deleted from  $R^{\mathcal{D}}$ , and the relationship  $\vec{r}/_{RN=\vec{e}}$  is inserted into  $R^{\mathcal{D}}$ .

2. The **reconnect** method is terminating. Moreover, **reconnect** is terminating with respect to any database instance that does not violate maximum cardinality constraints.

**Proof:** Termination is obtained immediately since **reconnect** is defined only in terms of terminating operations.

The **reconnect** update starts with the deletion of  $\vec{r}$  from  $R^{\mathcal{D}}$ . At this point, if  $R$  is non-binary, other relationships in  $R$  may become less consistent. If  $R$  is binary, the entities participating in  $\vec{r}$  may become less consistent. However, only the entity  $\vec{r}.RN$  has to be handled, since the other one is going to be reconnected soon. The compensation for the possible lose in minimal inconsistency of  $\vec{r}.RN$  has to be applied at this point since it might involve further deletions, that should not affect the insertions claimed by the **reconnect** application. So, for a binary  $R$ , and an entity  $\vec{r}.RN$  that is now minimally inconsistent, either the entity should be connected to another  $R$  relationship, or be deleted. Note that we do not wish to make the entity  $\vec{r}.RN$  consistent (since it might have been inconsistent in the beginning), but only to compensate for any lose in consistency. The **Nullify** and **Connect** steps bring the minimal inconsistency of  $\vec{r}.RN$  to its original level, and **Connect** takes care to make newly inserted entities consistent. The **Delete** step applies the *E.integrity\_delete* auxiliary update method to  $\vec{r}.RN$ . By Claim 3.8, the entity  $\vec{r}.RN$  is deleted without introducing new inconsistencies. Since this step is taken before the insertions of the entity identified by the key value  $A : v$  into  $(R.E\_of\_RN)^{\mathcal{D}}$  (if it is not already there), and the relationship  $\vec{r}_{/RN=\bar{e}}$  into  $R^{\mathcal{D}}$ , the possible entity and relationship deletions propagated by *E.integrity\_delete*, cannot spoil the claimed insertions.

Next, the entity  $\bar{e}$  identified by the key value  $A : v$  of  $E$  ( $E = R.E\_of\_RN$ ) is either retrieved or newly inserted. In the latter case, it may be inconsistent. Now the relationship  $\vec{r}_{/RN=\bar{e}}$  is connected to  $R$ . Since **connect** is cardinality faithful, no new inconsistencies are inserted, and existing ones can only decrease. Moreover, if  $R$  is binary, the entity  $\vec{r}.RN'$  (where  $R.role\_names = \{RN, RN'\}$ ) now gains back its lost relationship, so that its consistency is not affected any more. The following *E.make\_consistent*( $\bar{e}$ ) takes care of the possible inconsistency of the newly inserted  $\bar{e}$  (Claim A.1). So, at this point it is left only to take care of the inconsistencies caused by the deletion of  $\vec{r}$ , in case that  $R$  is non-binary. By Claim A.6, any direct or indirect increase in relationship inconsistency is compensated by the application of *R.make – deleted – relationship – consistent*( $\vec{r}$ ), without creating new inconsistencies. If during this last step the relationship  $\vec{r}_{/RN=\bar{e}}$  is deleted, the whole update is rejected. Hence, all claimed insertions and deletions are obtained and no new inconsistencies are created.  $\square$