Collection Type Constructors in Entity-Relationship Modeling

Sven Hartmann and Sebastian Link*

Information Science Research Centre, Massey University, New Zealand
{s.hartmann,s.link}@massey.ac.nz

Abstract. Collections play an important part in everyday life. Therefore, conceptual data models should support collection types to make data modeling as natural as possible for its users. Based on the fundamental properties of endorsing order and multiplicity of its elements we introduce the collection types of rankings, lists, sets and bags into the framework of Entity-Relationship modeling. This provides users with easy-to-use constructors that naturally model different kinds of collections. Moreover, we propose a transformation of extended ER schemata into relational database schemata. The transformation is intuitive and invertable introducing surrogate attributes that preserve the semantics of the collection. Furthermore, it is a proper extension to previous transformations, and results in a relational database schema that is in Inclusion Dependency Normal Form. In addition, we introduce a uniqueness constraint that identifies collections uniquely and guarantees referential integrity at the same time.

1 Introduction

The Entity-Relationship (ER) model has evolved into one of the most popular conceptual data models since its introduction in the late 1970s [5]. It is established as an excellent communication tool between systems analysts, database designers, managers and potential database users during the crucial process of identifying user information requirements. The ER model provides a well-defined semantics that is vital for a successful implementation of the information system under consideration, and offers modeling features that very much resemble the structure of natural languages [8].

Complex application domains, such as CAD/CAM, meta-modeling, hypermedia, office automation and life sciences, have resulted in the introduction of extended ER features [16,18]. These aim at providing natural modeling capabilities that adequately reflect complex object types that are inherent in everyday life activities. The most popular of these modeling constructors are the *tuple* and *cluster* constructor that represent aggregation and disjoint union [6,18]. Other kinds of very common complex objects are collections such as lists, sets, and

* This research is supported by the Marsden Fund Council from Government funding, administered by the Royal Society of New Zealand.

C. Parent et al. (Eds.): ER 2007, LNCS 4801, pp. 307–322, 2007.
© Springer-Verlag Berlin Heidelberg 2007

bags. Such constructors are present in many other data models including semantic data models [9], the nested relational data model [13], object-oriented data models [2], semi-structured data models [1] and XML [4], as well as sequence data models [11] and spatio-temporal data models [15]. In order to serve as a conceptual data model for these various approaches it is essential that the ER model supports these constructors in such a way that users are able to naturally take advantage of these extended capabilities. A further motivation to incorporate collection types into the framework of ER modeling is that they represent very often the modeling counterpart of *plurals* in natural languages [8].

Contributions. Since different types of collections do occur in everyday life conceptual data models should directly support the modeling of collections. Therefore, we introduce *collection type constructors* into the *formal* framework of Entity-Relationship modeling. Our constructors have a well-defined semantics and are very simple to utilise. Indeed, a collection type U has just a single object type component C that can model either finite lists, sets or bags of objects over C. The popularity of the ER model in the conceptual design phase is mainly due to its well-defined, intuitive and simple-to-use features. We strongly believe that our collection type constructors extend and enhance these characteristics further. In order to implement the conceptual design the conceptual database schema is usually mapped to a logical schema. The question is then how to map collection types and collections to object types and objects, respectively, of the logical data model. Our second objective is to address this question for the relational model of data which is still the de-facto standard model for most commercial database systems. While there are many different ways of implementing such a transformation our choice has several distinctive features: *i)* it extends the standard transformation of ER schemata [3,6,18] to relational database schemata to encompass collection types; *ii)* it is invertable preserving the semantics inherent in the collections at hand by introducing surrogate attributes that describe the membership relation of objects and collections, the position of an object within a list, and the multiplicity of an object within a bag; *iii)* the resulting relational database schema is in Inclusion Dependency Normal Form with respect to its functional and inclusion dependencies [10]. Finally, we propose an additional constraint that uniquely identifies collections within the relational database, and guarantees referential integrity at the same time.

Related Work. Complex-value databases have been studied extensively in the database literature [13], and we do not attempt to cite every work. It is therefore surprising that relatively little work has been done on extending the ER approach in this direction. Although nested attributes (multivalued attributes) have been introduced on the *attribute level* [14,16,18] the term collection type usually refers only to set-valued attributes. While tuple and cluster constructors are well-known features of ER modeling [18] collection constructors have not been properly investigated yet. This restricts the modeling capabilities unnecessarily, as we will demonstrate in this paper. Modeling collections has been discussed in other data models such as UML and ORM [7] but there is no provi-

sion of direct constructors for different collections. Instead, an annotation of the conceptual schema is proposed that guides the mapping process from conceptual to lower levels. In sharp contrast, our approach provides constructors for directly modeling collections, and our invertable transformation *automatically* encodes the structural properties of these collections into the relational database schema. It is therefore *different* from a simple flattening of complex structures. An alternative approach to modeling collections [17] encodes the structural properties of collections within an extensional uniqueness constraint. Our proposed transformation automatically derives a single constraint that identifies collections uniquely and guarantees referential integrity at the same time. Thus, the structural properties of collections are directly modeled on the conceptual model, and our transformation automatically encodes their distinctive properties on the logical level. It is therefore our strong belief that collection type constructors are very natural and useful features that increase the modeling capabilities of the ER approach without complicating the underlying theory.

2 Entity-Relationship Modeling

Since we intend to introduce new constructors into the ER framework, and since many different ER models do exist we will use this section to provide a common framework and to fix the semantics of the basic ER data model [5] and many of its extensions [3,6,18].

Entity Types. An *entity type* $E = (attr(E), id(E))$ consists of a name E, a finite and non-empty set of attributes $attr(E)$ such that each attribute $A \in attr(E)$ has a domain $dom(A)$, and a key $id(E) \subseteq attr(E)$ whose elements are called key attributes. An *entity* over E is a mapping $e : attr(E) \to \bigcup_{A \in attr(E)} dom(A)$ such that $e(A) \in dom(A)$ holds for all $A \in attr(E)$. An *entity set* E^t over E is a finite set of entities over E that satisfy the unique key value property, i.e., for all $e_1, e_2 \in E^t$ with $e_1(A) = e_2(A)$ for all $A \in id(E)$ we must have $e_1 = e_2$. We use $ent(E)$ to denote the set of all entities of an entity type E. We do not consider *weak* entity types due to their lack of a formal foundation and the problems caused by the identification of weak entities [18, pp.34-38].

Example 2.1. We can specify an entity type Client as follows: its attribute set is $attr$(Client) = {Name, Birthday, Address, Phone} with domain assignment dom(Name)=STRING, dom(Birthday)=DATE, dom(Address)=STRING, and dom(PHONE)=NUMBER. The key attributes of Client are id(Client) = {Name, Birthday}. An entity set may consist of the following three clients:

(John Fox, 08/08/1980, 88 Main Street, 3508888),
(John Fox, 02/12/1967, 23 Te Awe Awe, 3539465), and
(Lisa Hunter, 02/12/1967, 7 Park Ave, 356 1154).

Note that none of these clients has the same values on all key attributes. □

Relationship Types. A *relationship type* $R = (comp(R), attr(R), id(R))$ consists of a name R, a finite, non-empty set of components $comp(R)$, a finite set of attributes $attr(R)$ such that each attribute $A \in attr(R)$ has a domain $dom(A)$, and a key $id(R) \subseteq comp(R) \cup attr(R)$ whose elements are called key components and key attributes, respectively. A *relationship* over R is a mapping $r : comp(R) \cup attr(R) \to \bigcup_{E \in comp(R)} ent(E) \cup \bigcup_{A \in attr(E)} dom(A)$ such that $r(E) \in ent(E)$ for all $E \in comp(R)$ and $r(A) \in dom(A)$ for all $A \in attr(E)$. A *relationship set* R^t over R is a finite set of relationships over R that satisfy the unique key value property, i.e., for all $r_1, r_2 \in R^t$ with $r_1(X) = r_2(X)$ for all $X \in id(R)$ we must have $r_1 = r_2$. We refer to entity and relationship types jointly as *object types*. Note that we use *set semantics* to describe relationships [18]. At the moment, $comp(R)$ consists of entity types only. However, we will discuss other options for components shortly.

Example 2.2. Let COPY=({CopyNo,Title,Year,Director},{CopyNo}) be an entity type and let RENTAL=({CLIENT,COPY}, {RentalDay, DueDay},{COPY, RentalDay}) be a relationship type. The two relationships

((John Fox,08/08/1980,88 Main Street,3508888),(001.001,The Godfather,F.F.Coppola, 1972),04/01/2007,06/01/2007),
((John Fox,02/12/1967,23 Te Awe Awe,3539465),(001.002,The Godfather,F.F.Coppola, 1972),05/01/2007,06/01/2007).

form a relationship set over RENTAL. □

Relationship Types with Role Names. It may well occur that a relationship type must be used to model relationships between objects of the same component. In order to avoid confusion we associate distinct *role names* with the components. Role names can also be utilised to improve the readability of the ER diagram. As an example, we may specify the relationship type DESCENDANT with components $comp(\text{DESCENDANT}) = \{Child : \text{CLIENT}, Parent : \text{CLIENT}\}$, an empty attribute set and key $id(\text{DESCENDANT}) = comp(\text{DESCENDANT})$.

Specialisation and Generalisation. Sometimes, objects in the target of the database can be represented by more than just a single abstract concept. The idea to derive a *subtype* from a more general *supertype* is known as *specialisation.* A subtype inherits all features of its supertype, but often adds some new properties. A subtype U may be modelled as a unary relationship type whose single component is just its supertype C. Clearly, U may have some additional attributes, and we may use C as the key for U, i.e. $U = (\{C\}, attr(U), \{C\})$. Note that every object of type C gives rise to at most one object of type U.

Occasionally, it is desirable to model alternatives, e.g., having a relationship to various kinds of objects. The idea to derive a new abstract concept that is more general than several other abstract concepts is known as *generalisation.* A *cluster type* U consists of a finite, non-empty set $comp(U)$ of components $C_1, \ldots, C_n$, normally $n \geq 2$. We denote this cluster type by $C_1 \oplus \cdots \oplus C_n$. Clusters model

disjoint unions, i.e., an object set $\mathcal{I}(U)$ associated with a cluster type U is the disjoint union of its component's object sets $\mathcal{I}(U) = \bigcup_{i=1}^{n} \{(i, o) \mid o \in \mathcal{I}(C_i)\}$.

Higher-Order Relationship Types. So far, we have only allowed entity types to occur as components of relationship types. However, best practice suggests to allow also relationship types and cluster types to occur as components of another relationship type. For convenience, we call these kinds of types jointly *object types*. Entity types are object types without components while all other object types have at least one component. We need to ensure that the components of an object type are well-defined. For that we assign an order to each object type. Let U be an object type with component set $comp(U)$. The *order* of U is 0 if U is an entity type, and k if all components of U have order less than k and at least one of its components has order $k-1$. It is simple to extend the definitions of relationship and relationship set, correspondingly [18].

An *Entity-Relationship schema* (ER schema) is a finite set $\mathcal{S}$ of object types such that for each object type U in $\mathcal{S}$ and each of its components C or $p : C$ in $comp(U)$ we have that the object type C belongs to $\mathcal{S}$ as well. An *instance* $\mathcal{I}$ of an ER schema $\mathcal{S}$ assigns each object type U in $\mathcal{S}$ an object set $\mathcal{I}(\mathcal{U})$ such that for each relationship type or cluster type U in $\mathcal{S}$, for each of its components C or $p : C$, and for each object $o \in \mathcal{I}(U)$ we have that $o(C)$ or $o(p : C)$ belongs to $\mathcal{I}(C)$. An *Entity-Relationship diagram* (ER diagram) of an ER schema $\mathcal{S}$ is a directed graph with the elements of $\mathcal{S}$ as nodes, and with edges from a node U to a node C for all components $C \in comp(U)$, and edges from node U to node C labelled with p for all components $p : C \in comp(U)$. An example of an ER-diagram is given in Figure 1.

Nested Attributes. Due to lack of space we will not go into details concerning the treatment of nested attributes [14,16,18].

3 Syntax, Semantics and Examples of Collection Types

In database practice it becomes often desirable to model collections of objects. For example, a course might be taught by more than a single lecturer and the readings of this course may consist of a ranking of books. In such cases it is advantageous to have an abstract concept modeling a finite collection of objects of the same type. That is, we would like to derive a new object type from a given one by applying some kind of collection constructor. On the basis of endorsing an order between elements of a collection and/or multiplicity of elements within a collection we can naturally distinguish between four kinds of collections:

1. lists in which duplicates are allowed and order matters,
2. sets in which duplicates are not allowed and order does not matter,
3. bags in which duplicates are allowed and order does not matter, and
4. rankings (also known as ordered sets) in which duplicates are not allowed and order does matter.

A *list-,set-,bag-* or *ranking*-type U has a single component C, i.e. $comp(U) = \{C\}$. We use the following notation for collection types: double brackets allow duplicates while single brackets disallow duplicates. We write $\{\cdot\}$ to represent the absence of order, and $[\cdot]$ to represent its presence. Consequently, we denote a list type by $U[\![C]\!]$, a set type by $U\{C\}$, a bag type by $U\{\{C\}\}$ and a ranking type by $U[C]$, respectively. We write $U(C)$ to refer to a collection type without emphasising its particular type, i.e., $(\cdot)$ denotes one of the four collection brackets. The object set $\mathcal{I}(U)$ associated with a collection type $U(C)$ is just a set of finite lists (sets, bags, or rankings, respectively) of objects in $\mathcal{I}(C)$. The *key* of a collection type $U(C)$ is the collection type $U(C)$ itself: to identify any collection within a set $\mathcal{I}(U)$ of collections we need to know the collection. Collection types are visualised using a circle around the corresponding bracket and drawing a (labelled) edge to its component. Figure 1 shows an example. An *object type* may now refer to an entity, relationship, cluster or collection type, and an *object* to an entity, relationship or a collection. The definitions of higher-order relationship type, ER schema and ER diagram carry over. For an *instance* $\mathcal{I}$ of an ER schema $\mathcal{S}$ we add the requirement that for each collection type U in $\mathcal{S}$, and for its single component C or $p : C$, for each collection $O \in \mathcal{I}(U)$ we have that every $o \in O$ belongs to $\mathcal{I}(C)$. Note that this does not add any additional requirement for the empty collection in $\mathcal{I}(U)$.

Examples of Rankings. An object of a ranking type $U[C]$ is a finite ranking of C-objects, i.e., the C-objects in the ranking are totally orderded and the same C-object cannot occur more than once in the same ranking. Examples in which ranking types are useful modeling constructs can be found in everyday life. As a simple example consider an entity type WEBSITE with attributes *Name, Contents, URL* and *Size*. The key of WEBSITE is simply *URL*. The result of a web-search can then be modelled by a ranking of websites, i.e., we define the ranking type WEBSEARCH[WEBSITE].

Examples of Lists. An object of a list type $U[\![C]\!]$ is a finite list of C-objects, i.e., the C-objects in the list are totally orderded and the same C-object may occur repeatedly in the same list. A very simple example is a bit sequence in which we have an entity type BIT with a single (key) attribute *value* with domain $\{0, 1\}$. The list type SEQUENCE[[BIT]] models all finite lists of bits, i.e., $0, 1$-values.

Examples of Sets. An object of a set type $U\{C\}$ is a finite set of C-objects, i.e., the C-objects in the set occur precisely once and there is no order between them. The set type is significant whenever there is no preference between the elements and only the occurrences of distinct elements matter. For instance, one may be interested in the collection of all *distinct* articles a customer bought, or the collection of all students a professor has supervised. As a simple example we look at profiles of customers that purchase MP3s. In this particular case customers will not buy the same MP3 more than once (since they can copy it afterwards). Moreover, the order in which the customer selects the MP3s of a single purchase is not of interest. We may obtain the entity type MP3=($\{$*song,artist,album,genre*$\}$,$\{$*song,artist*$\}$), and the relationship

type PLAYER with component set {CUSTOMER,ORDER{MP3}}, attribute set {*day,price*} and key {CUSTOMER,ORDER{MP3},day}.

Examples of Bags. An object of a bag type $U\{\{C\}\}$ is a finite bag of C-objects, i.e., a C-object in the bag may occur repeatedly but there is no order between them. A simple example for using bag types are shopping profiles in which customers buy articles. The emphasis in this example is on the total price of the purchase. It therefore matters how many times the same article is purchased. We may use an entity type PRODUCT with attributes *p-ID, name, description* and *price* where *p-ID* forms the key, and a relationship type SHOPPING with components CUSTOMER and bag type BAG{{PRODUCT}}, attributes *time* and *type-of-payment*, and key {CUSTOMER,*time*}.

An Example. Consider the following ER schema of a university example:

- ACADEMIC=({Name, Phone, Office}, {Name}),
- BOOK=({ISBN, Price, Title}, {ISBN}),
- COURSE=({No, Title}, {No}),
- STAFF{ACADEMIC}, READINGS[BOOK]
- TEACHING=({COURSE,Lecturers:STAFF,Tutors:STAFF,READINGS},{Year},
  {COURSE,Lecturers:STAFF,Year})

Note that the key on TEACHING says that in every year the same course cannot be taught by the same set of lecturers. The corresponding ER diagram is illustrated in Figure 1.

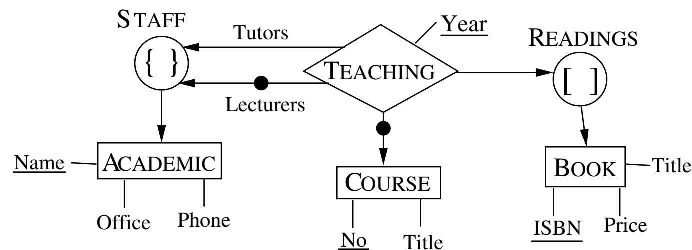

Fig. 1. ER diagram

4 Transformation to the Relational Model

An ER schema is the result of the conceptual design phase and serves as input for the following design steps, in particular the logical design phase. The framework of the latter is the relational model of data (RDM). Therefore, one needs to transform an ER schema with its abstract concepts into an RDM schema that uses flat relation schemata only. While the collection type constructors provide simple-to-use modeling features the transformation will actually reveal how these features can be implemented in relational tables. Our transformation does not simply flatten any nested structures but preserves the structural properties of collections, and makes therefore the mapping invertable.

4.1 Notions from the Relational Model of Data

In order to describe our transformation for collection types we will repeat fundamental notions from the RDM and summarise the transformation of object types different from collection types. This will also illustrate how neatly our extension fits into the current framework, and how the modeling capabilities extend without changing much of the underlying theory.

Let $\mathcal{D}$ denote a set of domains, i.e., a set of countably infinite sets. A *relation schema* R consists of a finite set $attr(R)$ of *attributes* and a *domain assignment* $dom : attr(R) \to \mathcal{D}$. A *tuple* over a relation schema R (for short: an *R-tuple*) is a mapping $t : attr(R) \to \bigcup_{D \in \mathcal{D}} D$ with $t(A) \in dom(A)$ for all $A \in attr(R)$. A *relation* over a relation schema R is a finite set r of R-tuples. For $X \subseteq R$ let $t[X]$ denote the projection of t on X. A *relational database schema* is a finite, non-empty set $\mathcal{S}$ of relation schemata. An *instance* $\mathcal{I}$ of a relational database schema $\mathcal{S}$ assigns to each $R \in \mathcal{S}$ a relation $\mathcal{I}(R)$ over R. A *key* on a relation schema R is a subset $K \subseteq attr(R)$ restricting relations r over R to satisfy $t_1[K] \neq t_2[K]$ for all $t_1, t_2 \in r$. A *key* K is called *minimal* if and only if no proper subset of K is a key. A *foreign key* on a relation schema R in a schema $\mathcal{S}$ is a sequence of attributes $A_1, \ldots, A_n \in attr(R)$ together with a minimal key $K = \{B_1, \ldots, B_n\}$ of some relation schema $S \in \mathcal{S}$ with $dom(A_i) = dom(B_i)$ $(i = 1, \ldots, n)$ restricting instances $\mathcal{I}$ of $\mathcal{S}$ to satisfy the inclusion dependency $R[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$, i.e., for each tuple $t \in \mathcal{I}(R)$ there must exist a tuple $t' \in \mathcal{I}(S)$ with $t[A_i] = t'[B_i]$ for all $i = 1, \ldots, n$. We use the notation $[A_1, \ldots, A_n] \subseteq S[B_1, \ldots, B_n]$ for a foreign key on R.

4.2 Transformation of Entity and Relationship Types

The transformation starts with entity types, and then continues with relationship types of order 1, then 2 and so on.

An entity type is simply just a different notation of a relation schema. The transformation is therefore very simple. The entity type $E = (attr(E), id(E))$ leads to a relation schema E' with $attr(E') = attr(E)$. The domain assignment for the attributes of E and E' is the same. Furthermore, E leads to a minimal key $id(E)$ on E'. Consider entity types as relationship types of order 0. For the entity types of the university example from Figure 1 we obtain

- Academic$'$ = {Name, Phone, Office} with minimal key {Name},
- Book$'$ = {ISBN, Price, Title} with minimal key {ISBN},
- Course$'$ = {No, Title} with minimal key {No}.

Suppose that we have transformed all relationship types of order n, and let $R = (comp(R), attr(R), id(R))$ be a relationship type of order $n + 1$. For each component $S \in comp(R)$ we choose pairwise disjoint sets

$$k_attr(S) = \{S.A \mid A \text{ key attribute of } S'\}$$

of new attribute names not occurring in $attr(R)$. Similarly, for a component $r : S \in comp(R)$ we choose $k_attr(r : S) = \{r.A \mid A \text{ key attribute of } S'\}$. The relationship type R results in the new relation schema R' with

$$attr(R') = \bigcup_{X \in comp(R)} k_attr(X) \cup attr(R).$$

For the domain assignment we have $dom(S.A) = dom(A)$, $dom(r.A) = dom(A)$, and $dom(A)$ remains unchanged for $A \in attr(R)$. Furthermore, R leads to the minimal key

$$\bigcup_{X \in id(R) \cap comp(R)} k_attr(X) \cup (id(R) \cap attr(R)) \text{ on } R'$$

and each component $S \in comp(R)$ defines a foreign key

$$[S.A_1, \ldots, S.A_n] \subseteq S'[A_1, \ldots, A_n]$$

on R'. In case of role names each component $r : S \in comp(R)$ defines a foreign key $[r.A_1, \ldots, r.A_n] \subseteq S'[A_1, \ldots, A_n]$ on R'. The transformation of an ER instance into its corresponding relational database instance is straightforward.

4.3 Transformation of Cluster Types

Clusters are used in conceptual design to model alternatives. The RDM does not provide a similar concept and one therefore transforms ER schemata with clusters into equivalent ER schemata without clusters [18]. This is only necessary as a pre-processing step before the actual transformation takes place. In general, clusters provide a convenient way to model objects in the target of the database. We do not recommend to avoid clusters as the size of an ER schema increases dramatically, and will therefore become much harder to comprehend.

4.4 Transformation of Collection Types

In the following we use U' to denote the name of the relation schema that corresponds to $U(C)$. For each collection type $U(C)$ the set $attr(U')$ of attributes of U' contains the set

$$k_attr(C) = \begin{cases} \{C.A \mid A \text{ is key attribute of } C'\}, & \text{if } C \text{ is not a collection type} \\ \{C'_{ID}\}, & \text{otherwise.} \end{cases}$$

and an additional surrogate attribute U'_{ID}. The attribute U'_{ID} identifies collections and, therefore, enables us to associate objects with collections in flat relations. The domain $dom(U'_{ID})$ is a set of surrogates.

Elements of bags may occur multiple times. Thus, a bag type $U\{\{C\}\}$ defines an additional attribute U'_{Mul} in $attr(U')$. It accommodates the information on the multiplicity of a bag's elements, and its domain is the set of positive integers.

Elements of rankings and lists have a position. Thus, a ranking or list type $U(C)$ defines an additional attribute U'_{Pos} in $attr(U')$. It accommodates the information on the position of an ordered collection's elements, and its domain is the set of non-negative integers.

In summary, for each collection type $U(C)$ in the ER schema, a relation schema U' with attributes $k_attr(C) \cup \{U'_{ID}\}$ is generated. For ordered collection types this relation schema will also contain an attribute U'_{Pos}, and for a bag type also U'_{Mul}. For the collection types of the example from Figure 1 we obtain

– $\text{Staff}' = \{\text{Academic.Name}, \text{Staff}'_{ID}\}$, and
– $\text{Readings}' = \{\text{Book.ISBN}, \text{Readings}'_{ID}, \text{Readings}'_{Pos}\}$.

Keys. An ordered collection type $U(C)$ defines the minimal key $\{U'_{ID}, U'_{\text{Pos}}\}$ on U'. That is, a tuple over U' is uniquely identified by the collection (the value in the U'_{ID}-column) and the position in this collection (the value in the U'_{Pos}-column). The difference between rankings and lists is that elements in a ranking uniquely determine the position within the ranking. In a list, however, the same element may occur in different positions. Thus, ranking types $U[C]$ result in the specification of a further minimal key, namely $k_attr(C) \cup \{U'_{ID}\}$ on U'.

A bag type $U\{\{C\}\}$ defines the minimal key $k_attr(C) \cup \{U'_{ID}\}$. That is, a tuple over U' is uniquely identified by the bag (the value in the U'_{ID}-column) and the values that identify the element of the bag (the values in the $k_attr(C)$-columns). In other words, every element of every bag has a fixed multiplicity.

A set type $U\{C\}$ defines the minimal key $attr(U')$. That is, a tuple over U' can only be uniquely identified by the set (the value in the U'_{ID}-column) and the values that identify the element of the set (the values in the $k_attr(C)$-columns). This is a consequence of the fact that the same element may occur in different sets, and different sets may have different elements.

In the university example, the minimal key on Staff' is Staff' itself, and the minimal keys on $\text{Readings}'$ are

$$\{\text{Readings}'_{ID}, \text{Readings}'_{Pos}\} \text{ and } \{\text{Book.ISBN}, \text{Readings}'_{ID}\}.$$

Foreign Keys. Let $U(C)$ denote an arbitrary collection type. If the object type C is not a collection type, then C defines a foreign key $[C.A_1, \ldots, C.A_n] \subseteq C'[A_1, \ldots, A_n]$ on U'. In the university example, the foreign key on Staff' is $[\text{Academic.Name}] \subseteq \text{Academic}'[\text{Name}]$, and the foreign key on $\text{Readings}'$ is $[\text{Book.ISBN}] \subseteq \text{Book}'[\text{ISBN}]$.

If C is the collection type $V(D)$, then $k_attr(C) = \{V'_{ID}\}$ and V'_{ID} uniquely identifies the collection but not the V'-tuple since the same V'_{ID}-value may be associated with different D-objects in the V'-relation (namely with all the elements of the collection V'_{ID} denotes). Consequently, we do not obtain a foreign key in this case. It is important to note at this point that we also do not specify the inclusion dependency $U'[V'_{ID}] \subseteq V'[V'_{ID}]$. The reason for this is the empty collection. Without the use of null values an empty collection of type $U(C)$ cannot be modelled in the U'-relation since empty collections do not have any elements. However, in the V'_{ID}-column of a U'-relation we may use a surrogate value to represent the empty collection. This surrogate value cannot occur in the V'_{ID}-column of the V'-relation. On the other hand, the use of the null value *not exists* for $k_attr(C)$-values permits the introduction of a U'_{ID}-value for the empty collection in the U'-relation. However, no attributes in $k_attr(C)$ can then be used as key attributes, and we would have to deal with incomplete information. Therefore, we prefer not to represent the empty collection in the U'-relation.

A Uniqueness Constraint. We will now introduce an additional constraint for those relation schemata that result from the transformation of collection types.

The purpose of the U'_{ID} attribute is to define the membership of elements in collections and to uniquely identify collections in the relational database instance. That is, any two U'_{ID}-entries in the database instance are the same precisely when they denote the same collection. In fact, if two different surrogate values from $dom(U'_{ID})$ occur in any relation, then these surrogate values denote different collections of type $U(C)$. That is, there must be a C-object t which separates the two collections **in the U'-relation**. Let $\mathcal{S}'$ denote the relational database schema obtained from transforming all object types of the ER schema $\mathcal{S}$. The active domain $adom_{\mathcal{S}'}(U'_{ID})$ is the union of all those values from $dom(U'_{ID})$ that occur in any U'_{ID}-column of an R-relation for any $R \in \mathcal{S}'$.

$$\begin{array}{c}\forall id_1, id_2 \in adom_{\mathcal{S}'}(U'_{ID}).(id_1 \neq id_2 \Rightarrow \exists t \in \prod_{A \in k_attr(C)} dom(A). \\ ((id_1, t) \in U'[U_{ID}, k_attr(C)] \wedge (id_2, t) \notin U'[U_{ID}, k_attr(C)]) \vee \\ ((id_1, t) \notin U'[U_{ID}, k_attr(C)] \wedge (id_2, t) \in U'[U_{ID}, k_attr(C)])).\end{array}$$

By abuse of notation $U'[X]$ denotes the projection of the U'-relation to the attributes in $X \subseteq U'$. Our uniqueness constraint really serves two purposes. Firstly, it guarantees that each collection can be identified uniquely by its surrogate from $dom(U'_{ID})$. Note that this is similar to the *extensional uniqueness constraint*, introduced by ter Hofstede et al. [17], which says that two sets are equal if and only if they have the same extension (i.e. the same elements). Secondly, our uniqueness constraint guarantees referential integrity over the RDM schema, i.e., it is a weak inclusion dependency saying that every element from $adom_{\mathcal{S}'}(U'_{ID})$ must also occur in the U'_{ID}-column of the U'-relation or denotes the empty collection. Notice that the constraint also eliminates the possibility that the empty collection is denoted by different surrogates. The uniqueness constraint is specified on the active domain of the U'_{ID} attribute in those U' that result from a collection type $U(C)$. In the university example we obtain the following uniqueness constraint on the active domain of Staff'_{ID}:

$$\begin{array}{c}\forall id_1, id_2 \in adom(\text{Staff}'_{ID}).(id_1 \neq id_2 \Rightarrow \exists t \in dom(\text{Academic.Name}). \\ ((id_1, t) \in \text{Staff}' \wedge (id_2, t) \notin \text{Staff}') \vee ((id_1, t) \notin \text{Staff}' \wedge (id_2, t) \in \text{Staff}')).\end{array}$$

Accordingly, we can define the uniqueness constraint for the active domain of $\text{Readings}'_{ID}$.

4.5 Object Types with Collection Type Components

The transformation described in Section 4.2 remains the same in the presence of collection types in the ER schema $\mathcal{S}$ taking into consideration the definition of $k_attr(X)$ from Section 4.4. The only difference is now that a collection type component U of the relationship type R does not define an inclusion dependency $R'[U'_{ID}] \subseteq U'[U'_{ID}]$ on R'. In fact, the unique surrogate value from $adom_{\mathcal{S}'}(U'_{ID})$ that may violate such an inclusion dependency is treated as if it denotes the corresponding empty collection. The relationship type TEACHING from the university example in Figure 1 results in the relation schema TEACHING$'$ with

– attributes COURSE.No, Lecturers:Staff$'_{ID}$, Tutors:Staff$'_{ID}$, Readings$'_{ID}$, Year,
– minimal key {COURSE.No, Lecturers:Staff$'_{ID}$, Year}, and
– foreign key [COURSE.No] ⊆ COURSE′[No].

Notice that we cannot specify any of the following inclusion dependencies:

– TEACHING′[Lecturers:Staff$'_{ID}$] ⊆ STAFF′[Staff$'_{ID}$],
– TEACHING′[Tutors:Staff$'_{ID}$] ⊆ STAFF′[Staff$'_{ID}$],
– TEACHING′[Readings$'_{ID}$] ⊆ READINGS′[Readings$'_{ID}$].

Rather than formalising the transformation of collections from the ER instance into flat relations we will use this section to illustrate this mapping by some examples. Therefore, consider the following, artificially small, ER database over our university schema.

COURSE	
No	Title
157266	Data Modeling
157357	IS Security

BOOK		
Title	ISBN	Price
ER Modeling	3540654704	138.-
DB Design	0201565234	90.-
Cryptography	0130914290	87.-
Viruses	0471007684	123.-

ACADEMIC		
Name	Office	Phone
Sven	2.09	7308
Sebastian	2.10	2717
Ernie	2.33	0077
Bert	2.33	0077

READINGS
[3540654704, 0201565234]
[0130914290, 0471007684]

STAFF
{Sven, Sebastian}
{Sebastian}
{Ernie,Bert}
∅

TEACHING				
COURSE	Year	READINGS	Lecturers:STAFF	Tutors:STAFF
157266	2007	[3540654704, 0201565234]	{Sven, Sebastian}	{Ernie, Bert}
157357	2007	[0130914290, 0471007684]	{Sebastian}	∅

This database will be transformed into the following relational database. We omit the representations of COURSE′, BOOK′ and ACADEMIC′ since these are just the same relations as the ones over COURSE, BOOK and ACADEMIC, respectively.

READINGS′		
ISBN	Readings$'_{ID}$	Readings$'_{Pos}$
3540654704	1	1
0201565234	1	2
0130914290	2	1
0471007684	2	2

STAFF′	
Name	Staff$'_{ID}$
Sven	2
Sebastian	2
Sebastian	1
Ernie	3
Bert	3

TEACHING′				
CourseNo	Year	Readings$'_{ID}$	Lecturers:Staff$'_{ID}$	Tutors:Staff$'_{ID}$
157266	2007	1	2	3
157357	2007	2	1	0

Notice how the nested STAFF-relation is represented as STAFF′-relation: every non-empty set S in the STAFF-relation is represented by a unique surrogate $n_S \in dom(\text{Staff}'_{ID})$, and for every element $e \in S$ the tuple $(e, n_S) \in dom(\text{Name}) \times dom(\text{Staff}'_{ID})$ represents this membership in the STAFF′-relation.

Notice that the active domain of Staff'_{ID} consists of $0, 1, 2, 3$ where 0 denotes the empty set. In fact, the value 0 is the unique surrogate that violates the inclusion dependency $\text{TEACHING}'[\text{Tutors:Staff}'_{ID}] \subseteq \text{STAFF}'[\text{Staff}'_{ID}]$.

4.6 Another Example for Mapping Collection Types and Collections

In order to illustrate the transformation for ER schemata in which collection types are nested we consider the following simple example of an ER schema:

WEBSITE=({URL,Name,Size},{URL}), and the two collection types SEARCH[WEBSITE] and MONITOR{{SEARCH}}.

Our transformation yields the following relational database schema

- WEBSITE′={URL,Name,Size} with minimal key {URL},
- $\text{SEARCH}' = \{\text{WEBSITE.URL}, \text{SEARCH}'_{ID}, \text{SEARCH}'_{\text{Pos}}\}$ with minimal keys $\{\text{SEARCH}'_{ID}, \text{SEARCH}'_{\text{Pos}}\}$ and $\{\text{WEBSITE.URL}, \text{SEARCH}'_{ID}\}$, and foreign key $[\text{WEBSITE.URL}] \subseteq \text{WEBSITE}'[\text{URL}]$
- $\text{MONITOR}' = \{\text{SEARCH}'_{ID}, \text{MONITOR}'_{ID}, \text{MONITOR}'_{\text{Mul}}\}$ with minimal key $\{\text{SEARCH}'_{ID}, \text{MONITOR}'_{ID}\}$.

For the database instance level we consider the following ER instance.

WEBSITE		
URL	Name	Size
http://www.sigmod.org/	SIGMOD Website	27.9 KB
http://www.acm.org/	ACM Website	26.75 KB
http://www.westwardlook.com/	Westward Website	Unknown

SEARCH
[http://www.sigmod.org/,http://www.acm.org/,http://www.westwardlook.com/]
[http://www.sigmod.org/,http://www.westwardlook.com/]
[ ]

MONITOR
{{ [http://www.sigmod.org/,http://www.acm.org/,http://www.westwardlook.com/], [ ], [ ], [ ], [http://www.sigmod.org/, http://www.westwardlook.com/], [http://www.sigmod.org/,http://www.westwardlook.com/] }}
{{ [http://www.sigmod.org/,http://www.westwardlook.com/], [http://www.sigmod.org/,http://www.westwardlook.com/], [http://www.sigmod.org/,http://www.westwardlook.com/] }}

This ER database can be represented as the following relational database. We omit the representation of WEBSITE′ since this is just the same relation as the one over WEBSITE.

SEARCH′		
WEBSITE.URL	SEARCH'_{ID}	$\text{SEARCH}'_{\text{Pos}}$
http://www.sigmod.org/	1	1
http://www.acm.org/	1	2
http://www.westwardlook.com/	1	3
http://www.sigmod.org/	2	1
http://www.westwardlook.com/	2	2

MONITOR′		
$\text{MONITOR}'_{ID}$	SEARCH'_{ID}	$\text{MONITOR}'_{\text{Mul}}$
1	0	3
1	1	1
1	2	2
2	2	3

Note that the value 0 in the SEARCH'_{ID}-column of the MONITOR′-relation denotes the empty ranking [ ] since it is not present in the SEARCH'_{ID}-column of the SEARCH′-relation. The active domain of SEARCH'_{ID} is $\{0, 1, 2\}$.

5 Properties of the Transformation

Our transformation enjoys several nice properties. Firstly, it is a proper extension of the standard transformation for ER schemata that only include relationship types and cluster types of any order [18].

Secondly, it preserves the semantics of its collection types, i.e., the values stored on the surrogate attributes that occur in a relation schema permit the reconstruction of the original collection in a straightforward manner. Basically, two objects belong to the same collection over $\mathcal{I}(U)$ precisely when they have been assigned the same U'_{ID}-value in the U'-relation. Moreover, the U'_{Pos}-value determines the position of an object in a ranking or a list. Finally, the U'_{Mul}-value denotes the multiplicity of an object within a bag. These properties make the transformation invertable. However, for a collection type $U(C)$ it cannot be decided whether the empty collection was an element of $\mathcal{I}(U)$ given the corresponding U'-relation. If $U(C)$ is the component of another object type O and a surrogate violates the inclusion dependency $O'[U'_{ID}] \subseteq U'[U'_{ID}]$, then the uniqueness constraint tells us that this surrogate represents the empty collection in the corresponding O'-relation. According to the definition of an ER instance the empty collection must have been an element of $\mathcal{I}(U)$. Mappings from relational databases to ER databases without collections have been studied previously [12].

Thirdly, some of the semantics of collections is reflected by keys on relation schemata that result from the transformation. For instance, since every C-object in a ranking over $U[C]$ determines its position within the ranking we obtain the functional dependency $\{U'_{ID}\} \cup k_attr(C) \rightarrow U'_{\text{Pos}}$, i.e., U'_{ID} and $k_attr(C)$ form a minimal key on U'. In lists, however, a C-object may occur in several positions of the list, but the list and the position within this list uniquely determine the C-object in this position (this is also true for rankings), i.e., $\{U'_{ID}, U'_{\text{Pos}}\}$

forms a minimal key on U'. Moreover, a bag and its C-object element together uniquely determine the multiplicity of the C-object, i.e., $\{U'_{ID}\} \cup k_attr(C)$ forms a minimal key for U'. In the special case of a set the multiplicity of an element is fixed to 1, and therefore there is no need for the surrogate attribute U'_{Mul} (this is a good example for a non-standard functional dependency, namely $\emptyset \rightarrow U'_{\text{Mul}}$).

Fourthly, the transformation results in a relational database schema that is in Inclusion-Dependency Normal Form with respect to the set of functional and inclusion dependencies obtained. Thus, it enjoys several desirable semantic properties [10] such as the absence of value and attribute redundancies and update anomalies. In fact, the transformation shows that the only functional dependencies defined on any of the resulting relation schemata are keys, and the inclusion dependencies are non-circular (due to the strict hierarchy of ER schemata) and key-based. It should be noted, however, that our uniqueness constraint implies some kind of weak inclusion dependency that is not key-based. It is future work to investigate the precise impact of the uniqueness constraints on design desiderata.

Finally, the transformation involving collection types does not result in a unique database instance since there are several choices for the values on the surrogate attributes. However, if the uniqueness constraint is satisfied, then it does not matter what these values are. Collections can be uniquely identified, and the relationship between different tables is guaranteed to be meaningful. In fact, every value occurring within the active domain of U'_{ID} either denotes the empty collection or it occurs in the relation over U' and references a unique collection. This results in a generalisation of the extensional uniqueness constraint introduced by ter Hofstede et al. [17].

6 Conclusion and Future Work

Entity-Relationship models use tuple- (aggregation) and cluster constructors to generate more complex object types from simpler ones. Collection types, however, have not been introduced into the formal framework of Entity-Relationship modeling. Previous work on this subject has either suggested to encode properties of collections into constraints [17] or annotate conceptual schemata and leave the burden of their implementation to lower design phases [7]. These solutions make it unnecessarily difficult for the designer to model collections naturally, and therefore to discuss the approximation of the target database with its users. This, in turn, defeats the purpose of a conceptual data model. In order to overcome these shortcomings we have introduced four collection type constructors into the ER framework. These have a well-defined semantics, and are intuitive and easy to use. The implementation of the collection types in relational tables can be done automatically by a transformation algorithm that enjoys many desirable properties.

In the future we would like to investigate the applicability of collection types to mappings into other data models such as XML [4] and sequence data models [11]. Several relational database management systems are now object-relational. It might be interesting to provide mapping algorithms that directly take into

account the collection types supported by such systems. It seems also desirable to provide query languages for the extended ER model, and to investigate integrity constraints in the presence of collection types.

References

1. Abiteboul, S., Buneman, P., Suciu, D.: Data on the Web: From Relations to Semistructured Data and XML. Morgan Kaufmann Publishers, San Francisco (2000)
2. Atkinson, M., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., Zdonik, S.: The object-oriented database system manifesto. In: Proceedings of the International Conference on Deductive and Object-Oriented Databases, pp. 40–57 (1989)
3. Batini, C., Ceri, S., Navathe, S.B.: Conceptual Database Design: An Entity-Relationship Approach. Benjamin Cummings (1992)
4. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0 (third edition) W3C recommendation 04 (february 2004), http://www.w3.org/TR/2004/REC-xml-20040204/
5. Chen, P.P.: The entity-relationship model: Towards a unified view of data. Transactions on Database Systems 1, 9–36 (1976)
6. Elmasri, R., Navathe, S.: Fundamentals of Database Systems, 4th edn. Addison-Wesley, London, UK (2003)
7. Halpin, T.: Modeling collections in UML and ORM. In: EMMSAD (2000), http://www.orm.net/pdf/EMMSAD2000.pdf
8. Hartmann, S., Link, S.: English sentence structures and EER modeling. In: The Fourth Asia-Pacific Conference on Conceptual Modelling. Conferences in Research and Practice in Information Technology, vol. 67, pp. 27–35 (2007)
9. Hull, R., King, R.: Semantic database modeling: Survey, applications and research issues. ACM Computing Surveys 19(3) (1987)
10. Levene, M., Vincent, M.: Justification for inclusion dependency normal form. IEEE Trans. Knowl. Data Eng. 12(2), 281–291 (2000)
11. Li, J., Ng, S., Wong, L.: Bioinformatics adventures in database research. In: ICDT. LNCS, vol. 2572, pp. 31–46. Springer, Heidelberg (2002)
12. Markowitz, V., Makowsky, J.: Identifying extended entity-relationship object structures in relational schemas. IEEE Trans. Softw. Eng. 16(8), 777–790 (1990)
13. Paredaens, J., De Bra, P., Gyssens, M., Van Gucht, D.: The Structure of the Relational Database Model. Springer, Heidelberg (1989)
14. Parent, C., Spaccapietra, S.: Complex objects modeling: An entity-relationship-approach. In: Nested Relations and Complex Objects. LNCS, vol. 361, pp. 272–296. Springer, Heidelberg (1987)
15. Parent, C., Spaccapietra, S., Zimányi, E.: Spatio-temporal conceptual models: Data structures + space + time. In: ACM-GIS, pp. 26–33 (1999)
16. Schek, H., Scholl, M.: The relational model with relation-valued attributes. Inf. Syst. 11(2), 137–147 (1986)
17. ter Hofstede, A., van der Weide, T.: Deriving identity from extensionality. International Journal of Software Engineering and Knowledge Engineering 8(2), 189–221 (1998)
18. Thalheim, B.: Entity-Relationship Modeling: Foundations of Database Technology. Springer, Heidelberg (2000)