Entity-Relationship Approach to Software Engineering
C.G. Davis, S. Jajodia, P.A. Ng and R.T. Yeh (eds.)
Elsevier Science Publishers B.V. (North-Holland)
© ER Institute, 1983

DESIGNING ENTITY-RELATIONSHIP SCHEMES FOR CONVENTIONAL INFORMATION SYSTEMS

Marco A. Casanova

Centro Científico de Brasília
IBM do Brasil
Caixa Postal 853
70.000, Brasília, DF
Brasil

José Eduardo Amaral de Sá
Instituto de Processamento de Dados e Informática da Marinha
Ilha das Cobras
20.000, Rio de Janeiro, RJ
Brasil

A methodology for designing entity-relationship schemas starting from the description of conventional information systems is proposed. It basically explains how to identify entity and relationship types by inspecting record types, and how to integrate the result into a conceptual schema. The methodology can be viewed as the fundamental step of the process of converting conventional systems to the database approach. Moreover, it can also be used to produce a high level documentation of such systems.

1. INTRODUCTION

We propose in this paper a methodology for designing entity-relationship schemas starting from the description of conventional information systems. We show how the objects of a conventional system - record types, files, data pools, etc... - can be interpreted into the basic concepts of the entity-relationship model [3]. Then, we transform the observations collected into a database design methodology.

The contributions of the paper are twofold. First, the conceptual schema design methodology proposed can be viewed as the fundamental step towards converting a conventional information system to the database approach. More precisely, we claim that the conversion process should start with the design of the conceptual schema and that the design should be based on the description of the record types, files and data pools. Then, the conversion process should continue with physical database design and the conversion of application programs up to the actual loading of the database. The advantages accrued from this approach are clear. First, the database is not designed from scratch, since we have chosen as the starting point the conventional system, which we implicitly assume to adequately model the real world application. We also avoided the naive approach of converting conventional files directly into database structures which, at best, would produce a loosely structured conceptual schema.

Second, the methodology may be used to provide a high-level documentation for conventional information systems based on the entity-relationship approach. This can be very helpful because, if we compare a conventional information system to the three level architecture proposed in [1], we observe that it can be easily identified with the internal schema, since it contains descriptions of stored records, field types and so on. However, it has long been argued that applications should be designed based on a higher-level description of the data, which is exactly the purpose of the conceptual schema.

It should be clear that our methodology covers just one step of the conversi process. It covers neither the complete design of the database, nor the conversi of application programs. It also leaves untouched the problem of initializing th database operation.

Most of the results in the paper also find an interpretation in terms of the relational model. Broadly speaking, the conversion process can be viewed as a method of mapping relational schemas into entity-relationship schemas. When the mapping is at all possible, we say that the relational schema is in entity-relationship normal form (ERNF). We justify this new normal form on the grounds that relational schemas in ERNF can be interpreted into higher-level concepts that hel understand the semantics of the database. ERNF also differs from previous normal forms [2] since it depends on the analysis of certain interrelational constraints called inclusion dependencies [4].

A methodology close to ours was developed in [8] for obtaining entity-relationship views of a relational conceptual schema. The similarities come from the fact that relation schemes and record types bear many points in common. However, our methodology is somewhat more complex because it must account for objects othe than record types. The CHARADE project [7] can also be compared to our methodolo at least in purpose.

This paper is organized as follows. Section 2 contains basic concepts and introduces the notation used. Section 3 investigates how record types and data pools may represent entity-relationship types. Section 4 describes a method of reducing the size of the original set of record types. Section 5 presents an algorithm that synthesizes entity-relationship schemas starting from conventional system descriptions. Section 6 contains remarks about the relational model that are related to the discussion in this paper and defines a new normal form. Finally, Section 7 contains conclusions and directions for future research.

2. BASIC CONCEPTS AND NOTATION

We assume that the reader is familiar with entity-relationship concepts, as described in [3]. We will also consider relationships of higher order, i.e., relationships involving other relationships. This extension also accomodates weak entities that get their identity from relationships of higher order. Besides being natural, this extension of the model simplifies considerably the discussion. So, we concentrate on the notation we will use to describe conventional files and records.

We assume in this paper that record types are flat, that is, they do not contain repeating groups.

The format of records in a conventional file will be described by a <u>record type</u>, which is a scheme of the form $R(A_1,...,A_n)$, where R is the <u>record name</u> and $A_1,...,A_n$ are <u>field names</u> (field types will be ignored in this paper). Sometimes we ignore field names and write that R is the record type.

A <u>record</u> of type $R(A_1,...,A_n)$ is simply a tuple r with length n. An entry of r is called a <u>field value</u>. If X is a subset of $A_1,...,A_n$ and r is a record of type R then $r(X)$ denotes the projection of r on X.

A <u>file</u> of type $R(A_1,...,A_n)$ is a set F of records of type R. Note that a file is simply a set with no structure whatsoever. Hence, this definition reflect the fact that the access method of a file, or even its ordering, if any, will play a marginal role in our discussion.

Let RS be a set of record types. A <u>state</u> of RS is a function v associati a file $F = v(R)$ to each record type R in RS such that all records in F are of type R.

Let RS be a set of record types. A <u>key</u> of a record type R in RS is a subset K of the set of field names of R. Given a state v of RS, we say that v <u>satisfies</u> K iff for any r and r' in $v(R)$, if $r(K) = r'(K)$ then $r = r'$.

From now on we assume that each record type description includes the defini-

tion of a set of keys over the record type.

In addition to keys, we will use another kind of constraints, borrowed from relational database theory. If $R(A_1,\ldots,A_n)$ and $R'(B_1,\ldots,B_m)$ are two record types in RS and X and Y are lists of field names of R and R', respectively, such that X and Y have the same length, then $R(X) \subseteq R'(Y)$ is called an inclusion dependency [4]. Given a state v of RS, we say that v satisfies $R(X) \subseteq R'(Y)$ iff for any record r in v(R) there is a record r' in v(R') such that $r(X) = r'(Y)$.

We will also use record expressions over RS defined just as relational expressions [11]. Given a state v of RS, we consider that v can be extended to expressions over RS in the usual way; the value of an expression E will be denoted by v(E).

We now introduce some concepts that are not standard, but which will play a central role in later sections.

Let RS be a set of record types and let $R(K) \subseteq S(L)$ be an inclusion dependency over RS such that L is a key of S. Then, we call $R(K) \subseteq S(L)$ a reference in RS and say that R references S via K and L. We also say that K is an out-key of R and that L is an in-key of S. We say that R references S iff there are K and L such that R references S via K and L.

Not all sets of record references will be considered since they do not lead to reasonable interpretations. Thus, we introduce the notion of a well-formed set of references.

Let RS be a set of record types together with their keys and let SIGMA be a set of references over RS. We say that SIGMA is well-formed iff for any two record types R and S in RS, if $R(K) \subseteq S(L)$ and $R(K') \subseteq S(L')$ are in SIGMA then $K = K'$ and $L = L'$. Thus, if SIGMA is well-formed, there is no ambiguity when we say that R references S, since R does so uniquely.

This concludes the list of basic definitions and notation we will need in the next sections.

3. MAPPING CONVENTIONAL OBJECTS INTO ENTITY-RELATIONSHIP CONCEPTS

We discuss in this section how the objects of a conventional information system may be mapped into entity-relationship concepts. The discussion will be informal with the help of an example.

In this and the next sections, we use just two kinds of constraints: keys and record references (i.e., dependencies of the form $R(X) \subseteq S(Y)$, where Y is a key of S). We assume that the description of each record type R includes the description of the keys of R, so that we explicitly introduce only the record references.

3.1 An Example

Consider a conventional system whose files are described by the following set of record types (call it RS):

(1)	DEPT(DNAME)	key DNAME
(2)	DEPT_MGR(DNAME,MGR)	key DNAME
(3)	CLERK(SSN,NAME,AGE)	key SSN
(4)	EMP(NO,POSITION,DNAME)	key NO
(5)	DEPENDENT(NO,NAME,DOCNM)	key NO,NAME
(6)	DOCTOR(DOCNM,SPECIALITY)	key DOCNM

Without further analysis, we cannot infer anything. In particular, we want to stress that the fact that DNAME is a field name of DEPT and DEPT_MGR is not a sufficient indication that any association exists between DEPT and DEPT_MGR. At most, we may admit that these fields have the same domain, if a careful naming convention was followed. (Certain view integration methodologies [9,10] would identify the two occurrences of DNAME without further considerations, which is contrary to our view [5]).

However, if we assume the following set of references in RS (call it SIGMA), certain inferences can be draw:

(7) DEPT_MGR(DNAME) $\subseteq$ DEPT(DNAME)
(8) DEPT_MGR(MGR) $\subseteq$ EMP(NO)
(9) CLERK(SSN) $\subseteq$ EMP(NO)
(10) EMP(NO) $\subseteq$ CLERK(SSN)
(11) EMP(DNAME) $\subseteq$ DEPT(DNAME)
(12) DEPENDENT(NO) $\subseteq$ EMP(NO)
(13) DEPENDENT(DOCNM) $\subseteq$ DOCTOR(DOCNM)

Let us consider DOCTOR first. This record type does not reference any other record type, but it is referenced by other record types. Thus, we may agree that records of type DOCTOR may be freely inserted, but not deleted, depending on how we treat the references to DOCTOR. In general, we consider that a record type R represents an entity type when condition 1 is satisfied.

condition 1: R does not reference other record types.

Similar observations also apply to DEPT. However, since DEPT has just one field name, we may also interpret DEPT as representing a domain definition. This is further reinforced if the file associated with DEPT is treated as a data pool, that is, a file of relatively stable data that is frequently referred by application programs, but seldon updated.

In general, a data pool D is used to validate input records and impose uniformity of reference accross files. It can be viewed as either defining a domain or an entity set. The choice must take into account at least the following facts:

(a) if we decide to view D as defining a domain, we cannot insert or delete entries in D; however, the schema will be somewhat simpler since it will contain one less entity type. The data pool D will usually have just one column when it defines a domain, although multicolumn domains are not impossible. The cardinality of D will also tend to be small.
(b) if, on the other hand, we decide to view D as defining an entity set, we are allowed to modify D; however, the schema will have one more entity type and, hence, will be more complex.

Another consequence is that the first choice leads to less relationship types in the schema than the second one.

This concludes our remarks about data pools.

Consider now DEPT_MGR. It references more than one record type and the set of out-keys contains a key. We then consider that DEPT_MGR defines a relationship type. In general, a record type defines a relationship type iff condition 2 holds where.

condition 2: R references more than a record type and the set of out-keys of R contains a key of R.

Note that we allow R to be referenced by other record types since we also consider here relationships of higher order.

Let us consider now EMP and CLERK. These two record types reference each other and, moreover, the references involve only keys. More precisely, we have that EMP(NO) is equal to CLERK(SSN) and that NO and SSN are keys of EMP and CLERK, respectively. So, each element in EMP(NO) is associated with exactly one record of type CLERK, and vice-versa. Thus, we may collapse EMP and CLERK into one record type.

(14) EMPLOYEE(NO,NAME,AGE,POSITION,DNAME) key NO

The references in (7) to (13) have to be modified adequately by replacing EMP and CLERK by EMPLOYEE as follows:

(15) DEPT_MGR(DNAME) $\subseteq$ DEPT(DNAME)
(16) DEPT_MGR(MGR) $\subseteq$ EMPLOYEE(NO)
(17) EMPLOYEE(DNAME) $\subseteq$ DEPT(DNAME)
(18) DEPENDENT(NO) $\subseteq$ EMPLOYEE(NO)
(19) DEPENDENT(DOCNM) $\subseteq$ DOCTOR(DOCNM)

Now, we consider that EMPLOYEE does not represent just an entity type since Condition 1 is not satisfied. Thus, we propose to break EMPLOYEE into two record types:

(20) EMPL(NO,NAME,AGE,POSITION) key NO
(21) EMPL_DEPT(NO,DNAME) key NO

and to modify SIGMA accordingly:

(22) DEPT_MGR(DNAME) $\subseteq$ DEPT(DNAME)
(23) DEPT_MGR(MGR) $\subseteq$ EMPL(NO)
(24) EMPL_DEPT(NO) $\subseteq$ EMPL(NO)
(25) EMPL_DEPT(DNAME) $\subseteq$ DEPT(DNAME)
(26) DEPENDENT(NO) $\subseteq$ EMPL(NO)
(27) DEPENDENT(DOCNM) $\subseteq$ DOCTOR(DOCNM)

Now, EMPL satisfies Condition 1 and, hence, we may consider that EMPL represents an entity type. As for EMPL_DEPT, it defines a relationship type, just like DEPT_MGR.

Finally, since EMPL and EMPL_DEPT were obtained by splitting EMPLOYEE, we may conclude that a single record type, EMPLOYEE in this case, may originate an entity type and a relationship type.

The analysis of DEPENDENT is also interesting. DEPENDENT references two other record types, but the union of its out-keys does not contain a key. Hence, DEPENDENT does not satisfy Condition 2. We then propose to break DEPENDENT into two record types:

(28) DEP(NO,NAME) key NO,NAME
(29) DEPDOC(NO,NAME,DOCNM) key NO,NAME,DOCNM

References (26) and (27) are replaced by:

(30) DEP(NO) $\subseteq$ EMPL(NO)
(31) DEPDOC(NO,NAME) $\subseteq$ DEP(NO,NAME)
(32) DEPDOC(DOCNM) $\subseteq$ DOCTOR(DOCNM)

Now DEPDOC represents a relationship type since it satisfies Condition 2. As for DEP, we will consider that it represents a weak entity type subordinated to EMPL, since it references EMPL via NO, which is a subset of its key. In general, we consider that a record type R represents a weak entity when condition 3 is met:

condition 3:R references just one record type and the out-key of R is contained in a key of R.

The final result of our discussion is a set of record types (call it RS'):

(32) DEPT(DNAME) key DNAME
(33) DOCTOR(DOCNM,SPECIALITY) key DOCNM
(34) EMPL(NO,NAME,AGE,POSITION) key NO
(35) DEP(NO,NAME) key NO,NAME

(36) DEPT_MGR(DNAME,MGR) key DNAME
(37) EMPL_DEPT(NO,DNAME) key NO,DNAME
(38) DEPDOC(NO,NAME,DOCNM) key NO,NAME,DOCNM

and a set of references (call it SIGMA'):

(39) DEP(NO) $\subseteq$ EMPL(NO)
(40) DEPT_MGR(MGR) $\subseteq$ EMPL(NO)
(41) DEP_MGR(DNAME) $\subseteq$ DEPT(DNAME)
(42) EMPL_DEPT(NO) $\subseteq$ EMPL(NO)
(43) EMPL_DEPT(DNAME) $\subseteq$ DEPT(DNAME)
(44) DEPDOC(NO,NAME) $\subseteq$ DEP(NO,NAME)
(45) DEPDOC(DOCNM) $\subseteq$ DOCTOR(DOCNM)

Each record type in RS' either represents an entity type, a relationship typ or a weak entity type. The entity-relationship diagram interpreting RS' is shown in Figure 3.1. Each object in the diagram has the same name and the same attribu tes as the corresponding record type. However, for the sake of simplicity, attri butes are omitted.

This concludes the analysis of our running example. Sections 4 and 5 will bring precision to the discussion in this section.

The Final Entity-Relationship Diagram

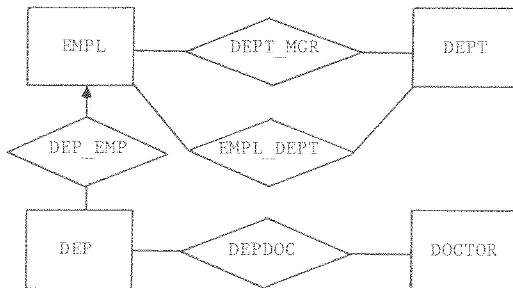

FIGURE 3.1

3.2 Typical Records in Conventional Systems

We analyse in this section, in terms of entity-relationship concepts, a kind of record type we claim to be commonly found in conventional systems. We hope that the discussion here will help understand the overall problem of designing entity-relationship schemas for conventional information systems.

A record type $R(A_1,\ldots,A_n)$ will frequently represent an entity type E, that we call base entity type, and as much as possible, n-1, binary relationships $R_1,\ldots,R_k$ involving E (on the "n" side). These relationships are called embedded.

Record types of the above form have many advantages:

(a) simplicity of design: grouping an entity type and several relationship types decrease the number of files:
(b) storage space economy: compressing many relationship types and an entity type in a single record avoids duplicating the entity key;

(c) search efficiency: the typical record type stores the natural join of E, $R_1,\ldots,R_k$, which simplifies searching files.

On the other hand, the use of such type of records tends to decrease:

(a) maintainability: a smaller number of files storing more information tends to induce complex programs that are more difficult to maintain;
(b) conceptual clarity: collapsing several relationship types and an entity type into a single record type tends to make the description less transparent;
(c) flexibility of design: a relationship type that is naturally n-m may be simplified to be n-1 or an n-m relationship may be inverted into several n-1 relationship types to conform to the format of such record types.

The above remarks can also be seen from a different point of view. The usua method of mapping entity-relationship schemas into relational schemas [3] can be directly used to produce conventional record types, instead of relation schemes. The method will produce a record type for each entity or relationship type. But the above observations suggest that it is profitable to produce a single record type from each entity type E together with all its binary n-1 relationships (with E on the "n" side).

4. FOLDING RECORD TYPES

We discuss in this section one of the problems raised by the example of Section 3.1, viz., that several record types may represent the same entity or relationship type. We characterize a situation in which this problem can be detected and define an operation, called folding, that collapses record types that represent tha same entity or relationship type into a single record type. Folding is essentially the operation we used to collapse EMP and CLERK into EMPLOYEE in Section 3.1.

The operation of folding is defined in three stages.

DEFINITION 4.1: Let RS be a set of record types and let SIGMA be a well-formed set of references over RS. The folding graph of SIGMA is the digraph G=(V,E) where:
(i) V = RS;
(ii) (R,S) is in E iff there is a reference $R(K) \subseteq S(L)$ in SIGMA where K is a key of R and L is a key of S. □

Note that, since we assumed that SIGMA is well-formed, to each arc (R,S) in E there corresponds a unique reference $R(K) \subseteq S(L)$ in SIGMA. We call $R(K) \subseteq S(L)$ the reference generating (R,S).

Whenever two record types in V belong to the same cycle in G, then they may potentially be folded together. The situation where folding is possible is captured in the following definition.

DEFINITION 4.2: Let G=(V,E) be the folding graph of SIGMA. Let R and S be two record types in V. We say that R and S are equivalent iff
(i) there is a cycle $(R_1,\ldots,R_k,R_1)$ in G such that R_1=R and R_j=S for some j in [1,k];
(ii) if $R_{m-1}(K_{m-1}) \subseteq R_m(K_m)$ and $R_m(L_m) \subseteq R_{m+1}(K_{m+1})$ generate (R_{m-1},R_m) and (R_m,R_{m+1}) , respectively, then $K_m = L_m$, for each m in [1,k] (sum and subtraction is module k).

We also say that K_m is the folding key of R_m and that the ith field of K_m corresponds to the ith field of K_j, for any m,j in [1,k]. □

DEFINITION 4.3: Let RS be a set of record types and let SIGMA be a well-formed set of references over RS. Let G=(V,E) be the folding graph of SIGMA. The folding of RS and SIGMA is the set RS' and SIGMA' of record types and references obtained as follows:

(1) let RS(1) = RS, SIGMA(1) = SIGMA and i=1, initially;

(2) suppose that there are two equivalent record types, R(X) and S(Y), in RS(i) with folding keys K and L:

(3) create RS(i+1) and SIGMA(i+1) as follows:

(3.1) RS(i+1) is RS(i) except that R(X) and S(Y) are replaced by R'(X') where X' contains each field in X or Y, except those in L;

(3.2) SIGMA(i+1) is SIGMA(i) except that:

- if $R(W) \subseteq T(V)$ (or $T(V) \subseteq R(W)$) occurs in SIGMA, replace it by $R'(W) \subseteq T(V)$ (or $T(V) \subseteq R'(W)$);
- if $S(W) \subseteq T(V)$ (or $T(V) \subseteq S(W)$) occurs in SIGMA, replace it by $R'(W') \subseteq T(V)$ (or $T(V) \subseteq R'(W')$) where W' is W with each field in L replaced by the corresponding field of K;

(4) increment i;

(5) repeat steps 2 to 4 until no new pair of equivalent record types can be found;

(6) RS' = RS(i) and SIGMA' = SIGMA(i), when the loop terminates. □

Folding has an important property that essentially says that no information is lost during the operation. This property is defined as follows:

DEFINITION 4.4: Let RS and RS' be sets of record types and let SIGMA and SIGMA' be sets of references over RS and RS', respectively. We say that RS' and SIGMA' represent RS and SIGMA iff we can associate an expression E_i over RS' with each record type R_i in RS such that, for each consistent state v' of RS', we have that v is a consistent state of RS, where v is such that $v(R_i) = v'(E_i)$. □

We can prove that the folding operation has the following property:

PROPOSITION 4.1: Let RS' and SIGMA' be the folding of RS and SIGMA. Then, RS' and SIGMA' represent RS and SIGMA, and vice-versa. □

This concludes the preliminary remarks we needed to define an algorithm that solves the ER-schema design problem.

5. DESIGNING ENTITY-RELATIONSHIP SCHEMAS

We transform in this section the remarks scattered in Section 3 into a methodology for designing an entity-relationship schema starting from a set of record types and a set of references.

We begin with a comprehensive definition of what it means for a record type to define an entity-relationship object in the presence of a set of references. These concepts were already illustrated in Section 3.

DEFINITION 5.1: Let RS be a set of record types and SIGMA be a well-formed set of references. Let R be a record type in RS.

(a) R contains an entity type iff R has a key L disjoint from all out-keys. We also say that L is an entity key.

(b) R contains a weak object type (i.e., a weak entity type or a weak relationsh type) iff R has an out-key K and a possibly empty set N of field names that do not belong to any out-key of R such that $L = K \cup N$ is a key of R. We also say that L is a weak object key and that R is subordinated to S via I if $R(K) \subseteq S(W)$ is the reference involving K.

(c) R contains a relationship type iff R has a set of out-keys $K_{i_1}, \ldots, K_{i_m}$, with $m > 1$, such that $L = K_{i_1} \cup \ldots \cup K_{i_m}$ is a key of R. We also say that L is a relationship key.

(d) R defines an entity type iff R does not reference any record type.

(e) R defines a weak object type iff R contains a weak object type and no entity type and no relationship type.

(f) R defines a relationship type iff R contains a relationship type and no entity type and no weak object type.

(g) R contains (defines) an ER-object iff R contains (defines) either an entity type, a weak object type or a relationship type. □

DEFINITION 5.2: Let RS be a set of record types and SIGMA be a well-formed set of references. We say that RS and SIGMA contain (define) an entity-relationship schema (or, simply, an ER schema) iff each record type in RS contains (defines) an ER-object. □

With the help of these definitions, we can precisely define the ER-schema design problem as follows: "Given a set RS of record types and a well-formed set SIGMA of references in RS such that the folding of RS and SIGMA contains an ER-schema, construct a set RS' of record types and a set SIGMA' of references over RS' such that: RS' and SIGMA' define an ER-schema; RS and SIGMA represent RS' and SIGMA'; RS' and SIGMA' represent RS and SIGMA.

We impose the condition that the folding of RS and SIGMA contains an ER-schema as otherwise the problem has no reasonable solution. Folding is necessary to collapse record types that represent the same entity or relationship type, thus avoiding trivial violations of the condition that RS and SIGMA contain an ER-schema

The first condition guarantees that we can associate an entity-relationship schema with RS' and SIGMA', which was the primary goal of this paper. The other two conditions guarantee that the schema has some very desirable properties. The second condition implies that we can construct a consistent state of RS' and SIGMA' from a consistent state of RS and SIGMA, which guarantees that the final stage of the conversion process - loading the database - can be correctly performed The last condition indicates that we can construct a consistent state of RS and SIGMA from a consistent state of RS' and SIGMA'. This condition is of ultimate importance because it implies that RS can be treated as a view of RS' and, hence, an application program does not initially have to be converted to the format of the database since it may run on an appropriate view of the schema.

Figure 5.1 describes an algorithm that solves the ER-schema design problem. The correctness of the algorithm in Figure 5.1 is proved in Theorem 5.1 below.

An Algorithm that Solves the ER-Schema Design Problem

```
ERDESIGN(RS,SIGMA;RS',SIGMA')
/*
   input : RS, SIGMA - a set of record types and a well-formed set of references
           over RS such that the folding of RS and SIGMA contains an ER-schema.
   output: RS', SIGMA' - a set of record types and references such that:
           (a) RS' and SIGMA' define an entity-relationship schema
           (b) RS' and SIGMA' represent RS and SIGMA
           (c) RS  and SIGMA  represent RS' and SIGMA'
   The following notation will be used:
           R(X)       - a record type in RS'
           P1,...,Pm  - the set of all keys of R
           F1,...,Fp  - the set of all record types such that there is a reference
                        R(Ki) ⊆ Fi(Mi) in SIGMA'
           T1,...,Tq  - the set of all record types such that there is a reference
                        Tj(Lj) ⊆ R(Nj) in SIGMA'
           Ki,Mi      - as above
           Lj,Nj      - as above
           M          - the set X - ((K1 ∪ ... ∪ Kp) - (P1 ∪ ... ∪ Pm))
*/
begin
(*)    fold RS and SIGMA into RS' and SIGMA';
(**)   for each record type R(X) in RS' that does not define an ER-object do
       begin /*
              RR0(M) defines the entity or weak object type contained in R and
                     has all keys of R and all references originally to R
           */
           delete R(X) from RS';
           add RR0(M) to RS' with the same keys as R;
           for each j = 1,...,q do
           begin delete Tj(Lj) ⊆ R(Nj)  from SIGMA';
                 add    Tj(Lj) ⊆ RR0(Nj) to  SIGMA';
           end
(***)      if R contains an entity type
              then let Pj be any entity key of R;
              else begin let Pj be any weak object key of R;
                         let Fs be the record type to which R is subordinated
                                to via Pj;
                         add RR0(Ks) ⊆ Fs(Ms) to SIGMA';
                   end
           /*
             RRi(Pj,Ki) defines a relationship type contained in R, i = 1,...,p
           */
           for each i = 1,...,p do
           begin add  RRi(Pj,Ki)  to  RS';
                 delete RRi(Ki) ⊆ Fi(Mi) from SIGMA';
                 add  RRi(Ki) ⊆ Fi(Mi)  to  SIGMA';
                 add  RRi(Pj) ⊆ RR0(Pj) to  SIGMA';
           end
       end
end
```

FIGURE 5.1

THEOREM 5.1: The algorithm in Figure 5.1 correctly solves the ER-schema design problem.

Proof

We have to prove that, given a set RS of record types and a well-formed set SIGMA of references over RS such that the folding of RS and SIGMA contains an ER-schema then the algorithm outputs a set RS' of record types and a set SIGMA' of references over RS' such that:

(A) RS' and SIGMA' define an ER-schema;
(B) RS' and SIGMA' represent RS and SIGMA;
(C) RS and SIGMA represent RS' and SIGMA';

We first prove (A). Since we assumed that the folding of RS and SIGMA contains an ER-schema, by Definition 5.2, Assertion (1) below holds after the folding operatior in line (*) of the algorithm:

(1) each record type in RS' contains an entity type, a weak object type or a relationship type.

We can easily check that (1) will be an invariant from then on.
The loop in line (**) must terminate because RS' initially contains a finite number of record types satisfying the loop condition and, at each iteration, each such record type is replaced by other record types not satisfying the loop condition. Indeed, let R be the record type selected at some loop iteration. Since R satisfies the loop condition, we must have:

(2) R defines neither an entity type, nor a weak object type nor a relationship type.

There are three cases to consider:

case 1: R contains an entity type.
Then, R is replaced in RS' by RR_0, that defines an entity type, and by $RR_1, \ldots,$ RR_p that define relationship types. So, R is replaced in RS' by record types not satisfying the loop condition.

case 2: R contains a weak object type, but no entity type.
Follows likewise.

case 3: R contains a relationship type, but no entity or weak object type.
Then, by Definition 5.1(f), R defines a relationship type, which contradicts (2). So, R could not have been selected at some loop iteration.

Therefore, we know that the loop terminates and, moreover, we know that the loop condition must be false in the final state. So, the loop terminates in a state satisfying:

(3) each record type R in RS' defines an ER-object.

From (3), we immediately establish (A). This concludes this part of the proof.

To prove (B) and (C), it suffices to observe that, by Proposition 4.1, (B) and (C) hold after the folding of RS and SIGMA and that (B) and (C) are preserved each time the algorithm replaces a record type R by other record types.

This concludes the proof.

To conclude, we observe that we can easily convert the output of the algorithm in Figure 5.1 into an entity-relationship schema by mapping each record type R into the ER-object E that it defines: the set of attributes and keys of E is equal to the set of field names and keys of R, respectively; the connections involving E are given by the references in SIGMA' involving R. Finally, we observe that E will be considered a relationship type or a weak object type of higher order if R references a record type R' that defines a relationship type.

6. REMARKS ON THE RELATIONAL MODEL

The discussion in previous sections, except about data pools, also applies to the relational model if we identify record types and relation schemes.

The goal of the algorithm in Figure 5.1 can then be rephrased as: given a relational schema, whose constraints are keys and inclusion dependencies of a special form (i.e., references), obtain an equivalent relational schema that defines an ER-schema. Note that this is just the inverse of the operation commonly considered, i.e., mapping ER-schemas into relational schemas.

We can go further and define a new normal form for relational schemas.

DEFINITION 6.1: Let S=(RS,C) be a relational schema, where RS is a set of relation schemes and C is a set of functional and inclusion dependencies.
We say that S is in Entity-Relationship Normal Form (ERNF) iff

(a) each relation scheme is in BCNF;
(b) each inclusion dependency $R[X] \subseteq S[Y]$ in C is such that Y is a key of S (we call it a reference);
(c) if $R[X] \subseteq S[Y]$ and $R[W] \subseteq S[V]$ are in C then X=W and Y=V;
(d) S defines an ER-schema (c.f. Definition 5.2). □

The advantages of having a schema in ERNF is that we can find an interpretation for the relation schemes in terms of entity-relationship concepts, which facilitates understanding the database.

We also note that ERNF intrinsically depends on interrelational dependencies, unlike most preceeding normal forms [2,6,11].

Finally, we observe that the algorithm in Figure 5.1 can be viewed as a normalization procedure that maps a relational schema that contains an ER-schema into an equivalent relational schema that defines an ER-schema.

7. CONCLUSIONS AND DIRECTIONS FOR FUTURE RESEARCH

We discussed, with the help of an example, how record types and data pools can be interpreted in terms of entity-relationship concepts. The analysis helped us understand how record types are designed and let to the development of a method for designing entity-relationship schemas from conventional information system descriptions.

The method depends on reasonable assumptions. However, since quite frequently record references are not explicitly stated, we may say that it has a chance of succeeding only if the conventional system has already been uniformized, data pools exist to harness cross-references between records from different files, and so on.

The design of the conceptual schema is just one of the steps necessary to convert conventional information systems to the database approach. The actual conversion (or not) of application programs and physical database design are certainly other important issues that must be investigated.

We have also left untouched the question of what dictionary facilities would help applying the method proposed.

REFERENCES

[1] Study Group on Data Base Management Systems. Interim Report, FDT 7,2, ACM (1975)

[2] Beeri, C. and Bernstein, P.A., A Sophisticates' Introduction to Database Normalization Theory, Proc. 4th Int'l Conf. on Very Large Data Bases (1978)

[3] Chen, P., The Entity-Relationship Model - Towards a Unified View of Data, ACM Trans. on Database Systems 1,1 (1976)

[4] Casanova, M.A., Fagin, R. and Papadimitriou, C.H., Inclusion Dependencies and Their Interaction with Functional Dependencies, Proc. 1st ACM SIGMOD/SIGACT Conf. on Principles of Database Systems (1982)

[5] Casanova, M.A. and Vidal, V.M.P., Towards a Sound View Integration Methodology, Proc. 2nd ACM SIGMOD/SIGACT Conf. on Principles of Database Systems (1983)

[6] Date, C.J., An Introduction to Database Systems (Addison-Wesley Pub.Co.,1981)

[7] Kent, W., Data Model Theory Meets a Practical Application, Proc. 7th Int'l Conf. on Very Large Data Bases (1981)

[8] Klug, A., Entity-Relationship Views over Uninterpreted Enterprise Schemas, Proc. 1st Int'l Conf. on the Entity-Relationship Approach to Systems Analysis and Design (1976)

[9] Navathe, S.B. and Gadgil, S.G., A Methodology for View Integration in Logical Database Design, Proc. 8th Int'l Conf. on Very Large Data Bases (1982)

[10] Teorey,T.J. and Fry, J.P., Design of Database Structures (Prentice-Hall, Inc. 1982)

[11] Ullman, J.D., Principles of Database Systems (Computer Science Press, 1979)