Information and Software Technology 42 (2000) 1057–1073

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.nl/locate/infsof

Enhanced ER to relational mapping and interrelational normalization

M. Kolp[a,*], E. Zimányi[b]

[a]University of Toronto, Department of Computer Science, 10 King's College Road, Toronto, Ont., Canada M5S3G4

[b]University of Brussels, Department of Informatics, Faculty of Engineering, 50 Av. F.D. Roosevelt, CP 165-15, 1050 Brussels, Belgium

Received 1 December 1999; revised 31 May 2000; accepted 31 May 2000

Abstract

This paper develops a method that maps an enhanced Entity-Relationship (ER+) schema into a relational schema and normalizes the latter into an inclusion normal form (IN-NF). Unlike classical normalization that concerns individual relations only, IN-NF takes interrelational redundancies into account and characterizes a relational database schema as a whole. The paper formalizes the sources of such interrelational redundancies in ER+ schemas and specifies the method to detect them. Also, we describe briefly a Prolog implementation of the method, developed in the context of a Computed-Aided Software Engineering shell and present a case study. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Inclusion normal form; ER+ model; Relational model; Normalization theory; Computer-aided software engineering tool

1. Introduction

Database design can be defined as the process of capturing the requirements of applications in a particular domain, mapping them onto a database management system, and tuning the implementation.

There is a general agreement on the division of database design into four steps [4,19,44]: requirements specification, conceptual design, logical design, and physical design. *Requirements specification* consists of eliciting requirements from users. *Conceptual design* develops requirements into a conceptual model (e.g. the ER+ model). The output of this step is called a *conceptual schema*. *Logical design translates* the conceptual schema into the data model (e.g. the relational model) supported by the target database management system. *Physical design* transforms the logical schema into a physical schema suitable for a specific configuration.

This paper deals with logical database design. Traditionally, this activity has been based on normalization of individual relations [5]. However, classical normalization cannot characterize a relational database as a whole. Thus, redundancies and update anomalies can still exist in a set of normalized relations. Two lesser known normal forms have been defined [20,21,30,31] to integrate the interaction of constraints in the database and detect redundancies.

Nowadays, relational database design typically goes through: first, conceptual (ER+) schema design; and second, translation into a relational schema. Conceptual models richer than the relational model provide a more precise and higher-level description of data requirements and constitute the starting point for logical design. Several methods have been proposed [19,20,21,33,42] for ER+-to-relational translations, but the semantic distance between the two models can lead to anomalies in the logical schema.

In order to propose a method for logical relational database design, we have taken into account such anomalies, especially redundancies detected by the new normal forms, and formalized their sources in the ER+ schema. We have improved the ER+-to-relational mapping and the database normalization rules given in Ref. [20] to take into consideration enhanced ER+ mechanisms [28].

Since database design is complex, a significant research development has been the adoption of knowledge-based techniques for automating design [40]. In the context of Computer-Aided Software Engineering (CASE) shell design, we have implemented in Prolog the algorithms, constructing a normalized relational schema from an ER+ one enhancing the proposals of Refs. [9,12,17,36].

The rest of the paper is structured as follows. Section 2 defines our version of the ER+ model and deals with basic relational concepts. It also introduces a running example, used throughout the paper. Section 3 is devoted to the normalization theory and introduces the new normal forms. Section 4 formalizes some sources of redundancy

* Corresponding author. Tel.: +1-416-978-7569; fax: +1-416-978-1455.
E-mail address: mkolp@cs.toronto.edu (M. Kolp).

0950-5849/00/$ - see front matter
PII: S0950-5849(00)00132-4

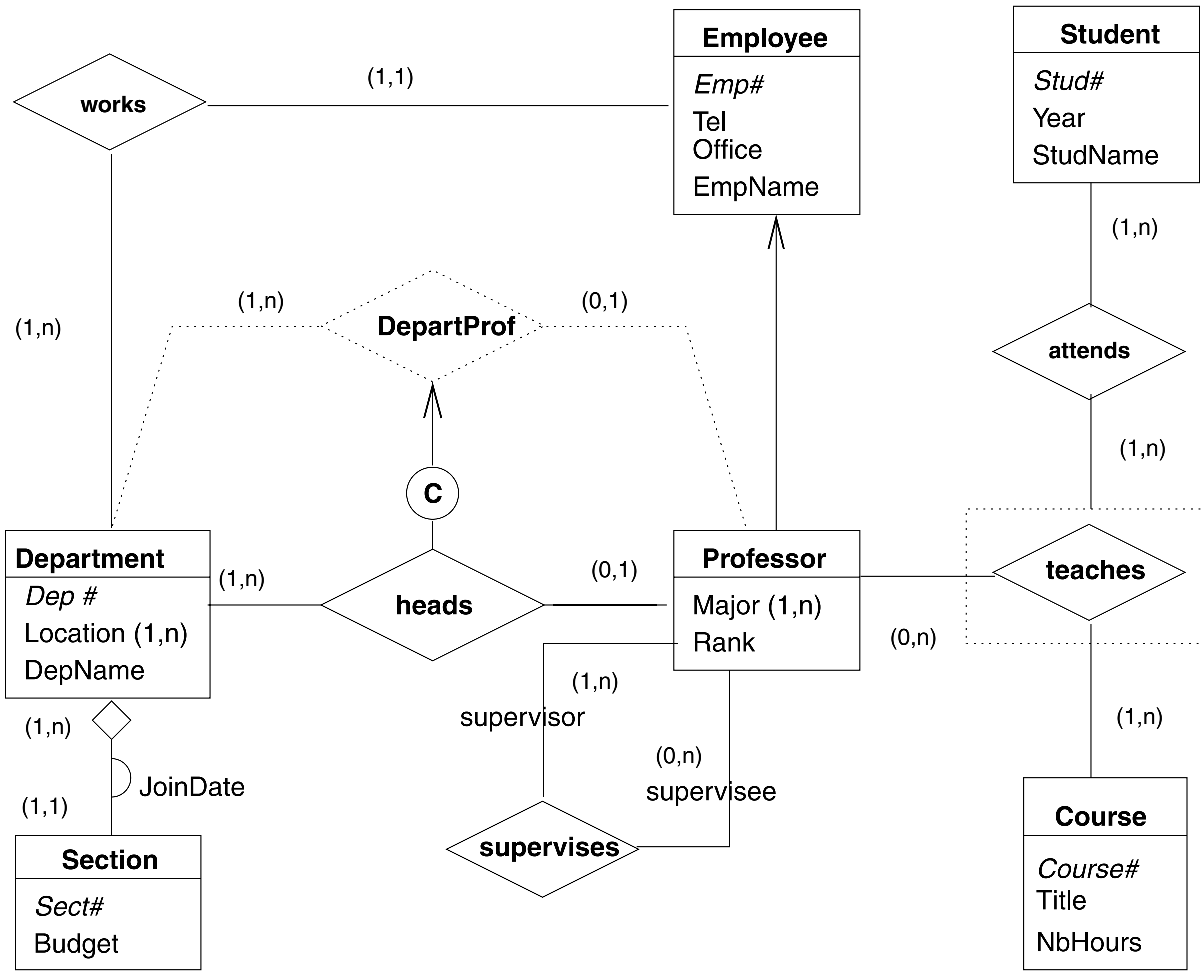

Fig. 1. An ER + schema.

in ER+ schemas detected by the inclusion normal form in the relational schema. Our enhancement of the design method based on Ref. [20] is explained in Section 5. Section 6 introduces the implementation of these mapping rules and the normalization algorithms in Prolog. We refer to Ref. [26] for a complete description of our implementation. A case study is briefly described in Section 7 to illustrate our approach. Related works are discussed in Section 8. Finally, Section 9 gives conclusions and pointers for further work.

2. ER+ and relational concepts

The ER model describes real world concepts with *entities* — objects of the application domain with independent existence — and *relationships* among entities. In the ER+ schema shown in Fig. 1, `Department` is an entity while `works` is a relationship. Each entity participating in a relationship is assigned one or more *roles*. Role names are omitted when there is no ambiguity. If an entity plays more than one role in a relationship, that relationship is said to be *recursive* and role names are mandatory. Fig. 1 shows a recursive relationship: `supervises` where `Professor` plays the roles of `supervisor` and `supervisee`.

Cardinality constraints model restrictions to relationships. In Fig. 1, every instance of `Department` participates in at least one and at most *n* (i.e. any number of) instances of `works`.

Attributes are properties of entities or relationships. For instance, `Employee` has an attribute `Emp#`. Attributes can be mono- or multi-valued. Thus, as for relationships, cardinalities are attached to attributes. The most frequent cardinalities are (1,1), which are assumed as default values and are omitted from the figures. In Fig. 1, `Department` has one and only one `Dep#`. On the contrary, `Location` is multivalued.

An attribute or combination of attributes of an entity is an *identifier* of the entity if its values, to exactly identify one instance of the entity. In Fig. 1, `Emp#` identifies `Employee`.

Several abstraction mechanisms have been added to the basic ER model. We consider weak entities, aggregated relationships, derived relationships, generalization, aggregation, and some additional explicit constraints.

A *weak entity* is an entity having no identifier of its own. Its instances are identified with respect to instances of one or more owner *entities*. A weak entity is connected to its owner entities via *identifying relationship*(*s*) and always has a (1,1) cardinality in these relationship(s). For example, Fig. 6 [4] shows three weak entities: DailyTrip, Segment, and DailySeg.

A weak entity usually has a *partial identifier*, which is the set of attributes uniquely identifying instances related to the same owner entity(ies). Thus, the identifier of the weak entity is the combination of an identifier of an owner entity and its partial identifier. In Fig. 6, an instance of DailyTrip is identified by the combination of Trip# and its partial identifier Date.

If a weak entity has no partial identifier, then it must define a set of identifying relationships that, when combined, uniquely identify weak entity instances. In Fig. 6, instances of DailySeg are identified by the combination of their owner entities Segment and DailyTrip, i.e. instances of DailySeg are identified by the combination of Trip#, Seg#, and Date.

Generalization is an abstraction mechanism involving two or more entities called, respectively, *superentity*(*ies*) and *subentity*(*es*). A superentity may have several subentities and vice versa. A superentity may also be a subentity of another superentity. A subentity inherits all the attributes and relationships of its superentities. Fig. 1 exhibits a generalization between subentity Professor and a superentity Employee.

Aggregation is an abstraction mechanism defining a *composite* entity from a set of (other) *component* entities. For example, in Fig. 1, the composite entity Department is composed of Sections. As for generalizations, composite and components entities may participate in other aggregations, leading to aggregation hierarchies. Similarly in relationships, cardinalities model restrictions between composites and components in aggregation hierarchies. Also, aggregations can have attributes, as shown by the attribute joinDate.

An *aggregated relationship* models a relationship as a *participant* in another relationship. For instance, the aggregated relationship teaches in Fig. 1 associates Student via attends with pairs of Professor and Course. Thus, in the rest of the paper, a participant in a relationship denotes an entity or an aggregated relationship.

A relationship is called *derived* [20,21,30,37] if it can be inferred from a combination (similar to a join) of other relationships and generalizations. Both paths (via the derived relationship and via the join of relationships or generalizations) represent the same association. In Fig. 1, DepartProf is a derived relationship.

A *subset constraint* models an inclusion constraint between two relationships. For instance, the subset constraint between heads and the derived relationship DepartProf models the constraint that every professor heading a department must work in that department and manage one project. Unlike Refs. [21,30], we differentiate generalization and subset constraints, since they correspond to different abstraction mechanisms.

Additional constraints can be defined on schemas: Office $\rightarrow$ Tel represents an FD (see below) meaning that, for every instance of Employee, the value of Office determines the value of Tel.

Fig. 2 shows a relational database schema obtained by applying an ER+-to-relational mapping to the ER+ schema of Fig. 1. A relation is associated to every entity and nonderived relationship, while a relational view is associated to every derived relationship. As will be shown later, this schema has redundancies and needs to be normalized. In particular, the relational view DepartProf, represented in dotted lines in Fig. 2, will allow the detection of these redundancies.

Data dependencies are constraints on databases and relations [44]. This paper only deals with functional and inclusion dependencies (FDs and INDs).

FDs are defined on individual relations and are represented, as in Fig. 2, by solid arrows. For instance, the FD Stud# $\rightarrow$ StudName on relation Student means that the value of Stud# determines the value of StudName. We sometimes denote an FD $X \rightarrow Y$ that holds on relation R by $R{:}X \rightarrow Y$.

INDs are interrelational constraints on pairs of relations represented, as in Fig. 2, by dashed arrows: the IND Attends[Stud#] $\subseteq$ Student[Stud#] means that the set of values of Stud# in Attends is a subset of the values of Stud# in Student. INDs involving keys are referred to as referential integrity constraints.

Given a set of data dependencies F, there are other FDs and INDs that also hold on a database satisfying the dependencies in F. The set of all such data dependencies is called the *closure* of F. It may be inferred by using inference rules.

Sound and complete sets of inference rules for FDs alone [1] and for INDs alone [10] are well known. Although there is no sound and complete set of inference rules for FDs and INDs taken together, the following rule is sound [10,34]:

Pullback Rule: If $R[XY] \subseteq S[WZ]$ and $W \rightarrow Z$, then $X \rightarrow Y$ with $|X| = |W|$.

The specification of real world constraints in a database schema constitutes an important part of conceptual design. Important integrity constraints are directly modeled by ER+ mechanisms, this is especially true for FDs and INDs. When the ER+ schema is mapped into a relational schema, these dependencies must be transferred to the corresponding relations.

An ER+ schema implicitly represents a set of FDs. For instance, an FD $Id(E) \rightarrow Y$ can be deduced from an entity E in an ER+ schema, if $Id(E)$ and Y are attributes of E and $Id(E)$ is an identifier of E. In Fig. 1, Stud# $\rightarrow$ StudName holds on entity Student; the corresponding constraint also holds in the corresponding relation in Fig. 2. In the same way, FDs explicitly represented in an ER+ schema also hold in the corresponding relations.

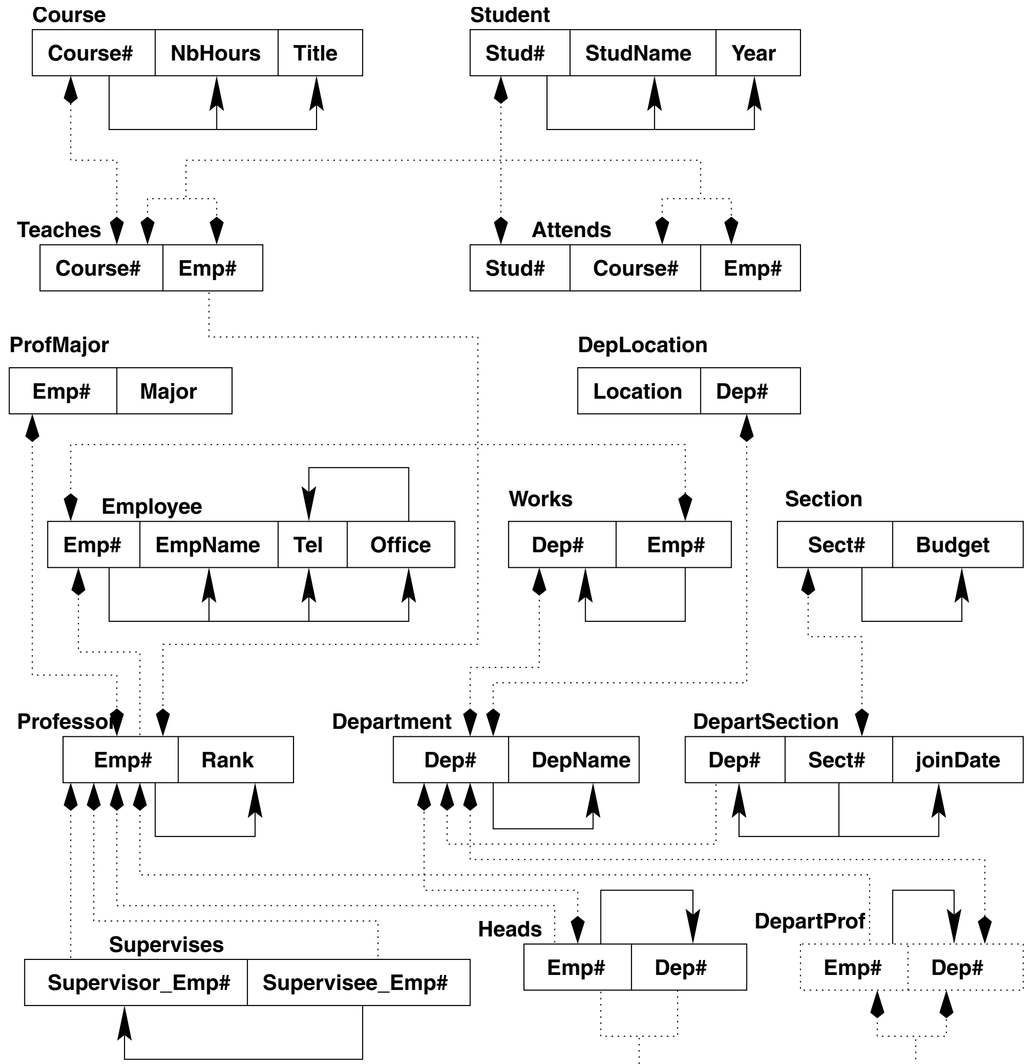

Fig. 2. A set of non normalized relations from the ER + schema of Fig. 1.

For relationships, an FD $Id(E_1) \rightarrow Id(E_2)$ can be inferred from a relationship R, if $Id(E_i)$ is an identifier of entity E_i and the maximal participation of E_1 in R is 1. In Fig. 1, an FD `Works: Emp#` $\rightarrow$ `Dep#` can thus be deduced.

Similarly, ER+ schemas implicitly model a set of INDs. For instance, an IND $R[Id(E)] \subseteq E[Id(E)]$ can be inferred from a relationship R and a participant E of R, where $Id(E)$ is the set of attributes of the identifier of E. In the example, the IND `Attends[Stud#]` $\subseteq$ `Student[Stud#]` implicitly holds.

Section 5.1 gives the mapping rules that deduce all implicit FDs and INDs in an ER+ schema and attach them to the corresponding relational schema.

3. Database normalization

Normalization [5,14,44] was introduced in relational database design to avoid redundancies, and to update anomalies due to data dependencies. This process is based on the application of normal forms to relations and databases. Each of these forms is specific to a type of data dependency. As already said, we only deal with normal forms concerning functional and inclusion dependencies.

The third normal form (3NF) guarantees individual relations without redundancies with respect to FDs. However, even if each relation is in 3NF, redundancies and update anomalies can still exist in a database considered as a whole due to INDs and to the interaction of FDs spanning several relations [2,11,21,30,31].

To circumvent these problems, the *Improved third normal form* (Improved 3NF) is introduced in Ref. [31]. Unlike classical normal forms, the Improved 3NF considers several relations rather than individual relations and determines redundancies with respect to FDs. Normalization into the Improved 3NF comprises the detection and deletion of superfluous attributes. It was proven that if a database is in the Improved 3NF, then each individual relation is in 3NF [31].

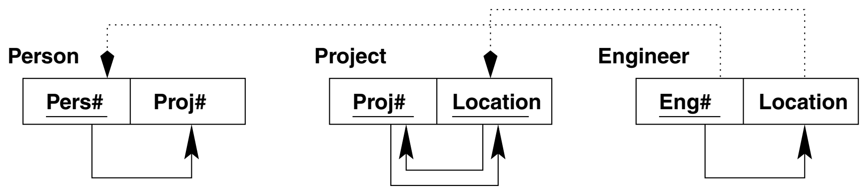

Fig. 3. A relational dataset not in IN-NF.

Inclusion normal form (IN-NF) [20,21,30] was later introduced to guarantee databases without redundancies with respect to FDs and INDs. It was also proven that if a database is in the IN-NF, it is also in the Improved 3NF [20,21,30].

As classical normalization theory concerns only individual relations, the choice of attribute names in different relations is not constrained. IN-NF and database normalization theory characterize a set of relations as a whole. Hence, we adopt a consequence of the Universal Relation Assumption [5]: if an attribute appears in two or more places in a database schema, then it refers to the same notion, for it represents the same semantics.

We now motivate the inclusion normal form with an example and then give its formal definition.

Fig. 3 shows a relational database, which is not in the IN-NF. Each person works on one project, each project is associated to one location and each engineer (who is also a person) is associated to one location. Suppose further that `Engineer[Eng#,Location]` $\subseteq$ `Person` $\bowtie$ `Project` `[Pers#,Location]` holds meaning that an engineer is located at the same place as the project on which she is working. Then, attribute `Location` in `Engineer` is said to be restorable since `Eng#` $\rightarrow$ `Location` can be deduced from the above IND and the FDs `Pers#` $\rightarrow$ `Proj#` and `Proj#` $\rightarrow$ `Location`. It is also said to be non essential since it is not needed to deduce other information (it is not part of any key of `Engineer`). Thus, it is superfluous and can be deleted as shown in Fig. 4. Note also that all dependencies involving `Location` in `Engineer` are removed.

Inclusion Normal Form: Consider a database D, a set Σ of FDs and INDs on D, a relation R and an attribute A of R. The dependencies in Σ not involving A in R, denoted $\Sigma_{R(A)}$, are the FDs $X \rightarrow Y \in \Sigma$, where $A \notin X$ and $A \notin Y$, as well as the INDs $R[X] \subseteq S[Y] \in \Sigma$, where S is not a relational view derived by join and projection from relations of D such that attribute A of R is necessary to perform a join in the construction of S.

A is *restorable* in R if its values can be deduced from $\Sigma_{R(A)}$. More precisely, A is restorable if there exists a key K of R not containing A, and such that we can infer the FD $K \rightarrow A$ from $\Sigma_{R(A)}$.

A is *non essential* in R if A is not necessary to deduce any other attribute of R. Formally, A is non essential in R if, whenever a key K of R contains A, there exists another key K' in R not containing A such that we can infer $K \rightarrow K'$ from $\Sigma_{R(A)}$.

A is *superfluous* in R if it is both restorable and non essential.

A database D is in IN-NF if there are no superfluous attributes in any relation schema of D.

The main difference between the Improved 3NF and the IN-NF is that, for inferring an FD $X \rightarrow Y$, the Improved 3NF considers only FDs while the IN-NF considers both the FDs and the INDs.

4. Relational redundancies implied by ER schemas

In this section, we consider superfluous attributes and relations and relate these redundancies with the corresponding ER schemas.

IN-NF is the only normal form taking into account relational redundancies relative to inclusion constraints. As shown in Section 2, such constraints can be modeled in ER+ schemas using subset constraints and derived relationships. For instance, Fig. 1 includes the inclusion constraint that every `Professor` heading a `Department` must `work` in that `Department`. This can be written, as follows, in a language based on logic:

$\forall p$: Professor, $\forall d$: Department (heads(p,d) $\Rightarrow \exists e$: Employee (isa(p,e) $\wedge$ works (e,d)))

For this logical constraint, a subset constraint models the implication — the inclusion constraint — while a derived

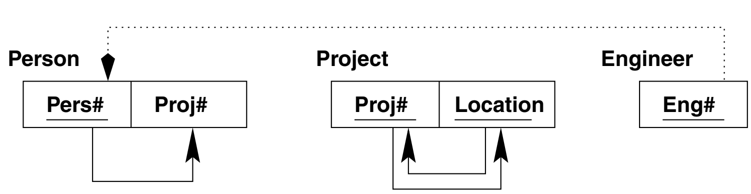

Fig. 4. Example of Fig. 3 normalized in IN-NF.

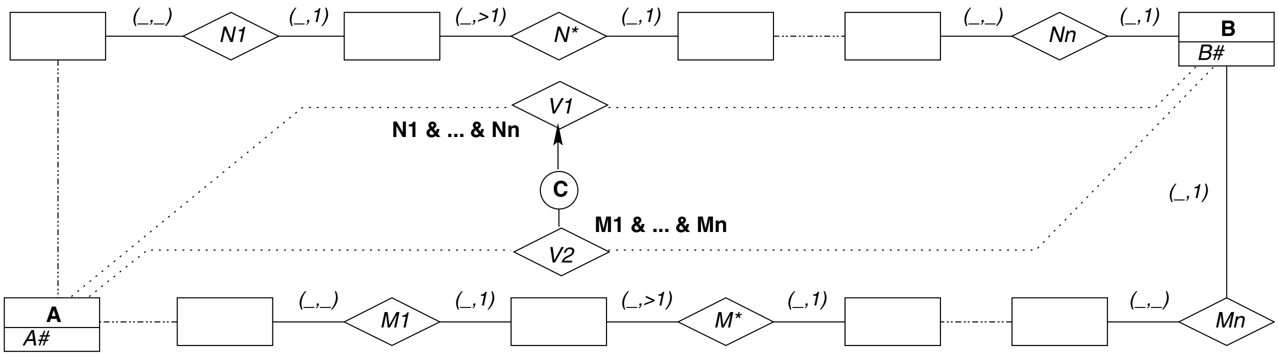

Fig. 5. A source of superfluous attributes.

relationship represents the conjunction of predicates — the association of other relationships.

Since inclusion constraints are often associated with ER cycles, they become possible sources of superfluous attributes in the corresponding relational schemas. In cycles, some information can be deduced in more than one way, which is the intuition of restorability: in Fig. 1, for a `Professor` who heads a `Department`, we can deduce the `Department` in which he works either via `heads` or via `works`. Moreover, some information is not necessary to deduce other information, which is the intuition of non essentiality: in Fig. 1, a `Department` can be headed by more than one `Professor`. Consequently, there is no constraint on `heads` by which a particular `Department` determines one and only one `Professor`.

While in the ER+ schema `heads` capture some important semantics of the real world (a `Professor` can head a `Department`), in the relational schema of Fig. 2 `Dep#` in `Heads` is restorable since we can deduce `Heads: Emp#` $\rightarrow$ `Dep#` from `Heads[Dep#,Emp#]` $\subseteq$ `DepartProf [Dep#,Emp#]` and from `DepartProf: Emp#` $\rightarrow$ `Dep#`. It is also non essential since it is not part of the left-hand side of any FD that holds on `Heads`. Therefore, `Dep#` should be removed from relation `Heads`.

On the contrary, if the cardinality between `Department` and `heads` is (1,1) instead of (1,n), then `Dep#` in `Heads` becomes essential: the FD `Heads: Dep#` $\rightarrow$ `Emp#` should hold and cannot be inferred from all dependencies not involving `Dep#` in `Heads`.

ER schemas are sources of superflous attributes, if they include cycles with an inclusion constraint of the kind shown in Fig. 5. Two entities `A` and `B` with identifiers `A#` and `B#` are related via, on the one hand, relationships M_i and, on the other hand, relationships N_i. All M_i and N_i, except M^* and N^*, are either 1:1 or N:1 relationships in the `B` to `A` direction, and M^* and N^* are mandatory N:1 relationships in the same direction. V_1 and V_2 are derived relationships associating, respectively, relationships N_i and M_i. The derived relationships and subset constraint model an inclusion constraint.

As already mentioned in Section 2, an FD $Id(E_1) \rightarrow Id(E_2)$ can be inferred from a relationship R if $Id(E_i)$ is an identifier of entity E_i and the maximal participation of E_1 in R is 1. Consequently, we can generate, by transitivity on the corresponding relational schema, an FD `B#` $\rightarrow$ `A#` holding on a view mapping V_1 because of the (_,1) cardinality of each N_i. In the same way, the FD `B#` $\rightarrow$ `A#` also holds on a view mapping this time V_2. Thus `A#` is restorable.

Since we cannot generate the inverse FD `A#` $\rightarrow$ `B#`, because of the (_, > 1) cardinalities of M^* and N^*, `A#` is non essential and then superfluous in V_2.

We consider now redundant relations. In Fig. 2, relation `Heads` (now with only `Emp#`) is not redundant because it contains the subset of professors heading a department while `Professor` contains all professors. This is represented by the IND `Heads[Emp#]` $\subseteq$ `Professor [Emp#]`.

Suppose now that cardinality between `Professor` and `heads` is (1,1) instead of (0,1), meaning that all professors head a department. Now, since the participation of `Professor` is mandatory in relationship `heads`, the inverse IND `Professor[Emp#]` $\subseteq$ `Heads[Emp#]` holds. Consequently, relation `Heads` is redundant since all information contained in `Heads` is also contained in `Professor`: relation `Heads` should then be deleted.

Similarly, suppose that `Professor` only has the multivalued attribute `major`. Then, relation `Professor` in Fig. 2 should also be deleted because it is redundant with respect to `ProfMajor`. Both `ProfMajor[Emp#]` $\subseteq$ `Professor [Emp#]` and `Professor[Emp#]` $\subseteq$ `Prof Major[Emp#]` hold meaning that all information contained in `Professor` is also represented in `Prof Major`.

Notice also that all attributes of `Teaches` are included in relation `Attends`. Since both INDS `Teaches [Course#, Emp#]` $\subseteq$ `Attends [Course#, Emp#]` and `Attends [Course#, Emp#]` $\subseteq$ `Teaches [Course#, Emp#]` hold, then relation `Teaches` is redundant and should be removed. On the contrary, if relation `Teaches` had another attribute, such as `semester` not present in `Attends`, then it would not be redundant.

5. Design method

Traditionally, database design has been accomplished

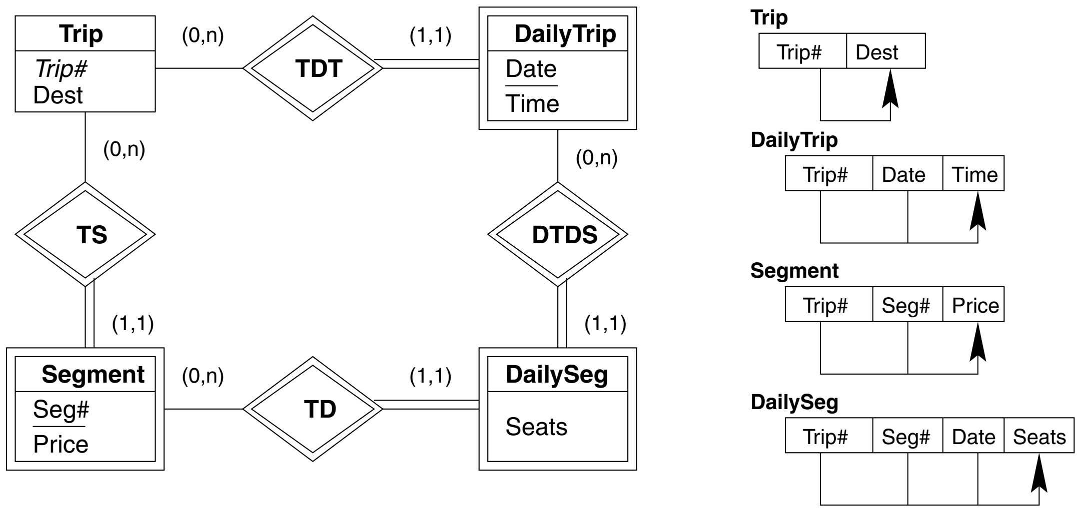

Fig. 6. Mapping weak entities into relations.

using normalization. However, since the adoption of conceptual models in the mid-1970s, normalization theory ceased to be the main logical design step. In fact, working first with ER or another rich conceptual model directly produces 3NF relations in most cases.

Nowadays, normalization is only viewed as a verification step removing anomalies left by the ER-to-relational mapping. However, usual ER-based design methods remain focused on attaining classical normal forms (3NF or BCNF) without removing other kinds of redundancies studied in this paper. As shown in Section 4, ER cycles can be sources of superfluous attributes not detected by classical normalization. Hence, the interest of enhanced ER-based design methods that remove anomalies due to cycles and inclusion constraints.

We propose an integrated design method including normalization into IN-NF based on Ref. [20]. These algorithms comprise three main steps. The following sections develop these steps.

1. ER+-to-relational mapping (see also Refs. [19,20,21,33,42,44]): an ER+ schema is mapped into a set of non normalized relations. Data dependencies are generated to represent implicit constraints of the ER+ schema.
2. Relation normalization and key generation: each relation is decomposed into a set of 3NF relations and at least one key is found for each 3NF relation by using the Bernstein algorithm [6]. Since several papers present this algorithm and its implementation in detail (see e.g. Ref. [12]), we do not develop this phase in the paper.
3. Database normalization: superfluous attributes and relations are deleted lending a database in IN-NF.

5.1. ER+-to-relational mapping

Entities: Each entity E is mapped into a non normalized relation of the same name, comprising all single valued attributes of E. For instance, entity `Student` in Fig.1 is mapped into relation `Student` in Fig. 2. As said in Section 2, FDs implicitly present in an entity E hold on the relation representing E. In the example, `Student: Stud#` → `StudName` is generated.

Similarly, FDs explicitly added to an entity E also hold on the relation representing E. In our example, `Employee: Office` → `Tel` also holds.

Weak entities: Given a weak entity W and its identifying entities $I_1,\ldots,I_n$, add to the relation R representing W the identifiers $Id_1,\ldots,Id_n$, where Id_i is an identifier of I_i. Consider the example shown in Fig. 6.

Each weak entity has either a partial identifier (as in `Segment` and `DailyTrip`) or can be identified by a combination of its identifying entities (as in `DailySeg`). For the first case, `Trip#` is added to both relations `Segment` and `DailyTrip` and thus, the identifiers of `DailyTrip` and `Segment` are, respectively, `Trip# Date` and `Trip# Seg#`.

On the other hand, `DailySeg` has no partial identifier but can be identified by a combination of its identifying entities `DailyTrip` and `Segment`. Therefore, the union of the identifiers of both entities, i.e. `Trip#`, `Seg#`, and `Date`, must be added to relation `DailySeg`.

Furthermore, an FD R: $X \rightarrow Y$ holds, if Y belongs to R and either: (1) X is the combination of Id_i and the partial identifier of W, or (2) X is the combination of the identifiers $Id_{i_1},\ldots,Id_{i_j}$ of its identifying entities. For the first case, the FDs `DailyTrip: Trip# Date` → `Time` and `Segment: Trip# Seg#` → `Price` hold. For the second case, `Trip# Seg# Date` is the identifier of `DailySeg` and the FD `DailySeg: Trip# Seg# Date` → `seats` holds.

Generalizations: Given a subclass E and its direct superclasses $S_1,\ldots,S_n$, add to the relation R representing E one of the identifiers $Id(S_i)$ of S_i for each i. In our running example, relation `Professor` inherits `Emp#` from `Employee`.

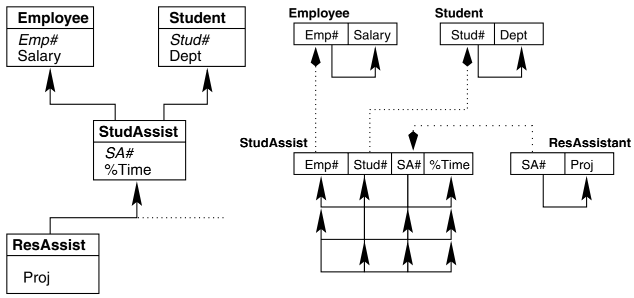

Fig. 7. Mapping multiple inheritance.

Now, consider Fig. 7 to illustrate multiple inheritance. Since `StudAssist` is a direct subclass of both `Employee` and `Student`, `Emp#` and `Stud#` must be added to relation `StudAssist`. Thus, all (own or inherited) attributes of `StudAssist` can be obtained through natural joins on `Emp#` and `Stud#`. `StudAssist` has three identifiers but only one of them is needed to be inherited to relation `ResAssist` to be able to access all its own or inherited attributes. In the example, `SA#` is chosen.

Further, an FD R: $Id(S_i) \rightarrow Y$ holds, if Y belongs to R. For instance, the FD `Professor:Emp#` $\rightarrow$ `Rank` holds in Fig. 2.

An IND $E[Id(S)] \subseteq S[Id(S)]$ is generated from a subclass E and one of its direct superclasses S, where $Id(S)$ is the common identifier of E and S. In our example, the IND `Professor[Emp#]` $\subseteq$ `Employee[Emp#]` holds.

Relationships: Each non derived relationship R is mapped into a non normalized relation with the same name, comprising all single valued attributes of R and the identifiers of R, i.e. the identifiers of all entities participating in R directly or indirectly through aggregated relationships. Consider a relationship R and a set of participants (entities or aggregated relationships) $E_1,\ldots,E_n$ of R. The identifier of R, $Id(R)$, is recursively defined by $Id(R) = Id(E_1) \cup \ldots \cup Id(E_n)$, where $Id(E_i)$ is as follows:

- if E_i is an entity, then $Id(E_i)$ is the set of attributes of the identifier of E_i;
- otherwise, E_i is an aggregated relationship over $A_1,\ldots,A_n$ then $Id(E_i) = Id(A_1) \cup \ldots \cup Id(A_n)$.

In the presence of FDs, the identifier $Id(R)$ of a relationship, as defined above, may not be minimal and thus some attributes may still be removed, as will be explained shortly.

In Fig. 2, the identifier of relation `Teaches` (where every participant is an entity) is `{Course#, Emp#}`. On the other hand, the identifier of relation `Attends` is `{Stud#,Course#,Emp#}`, i.e. the identifier of `Student` and the identifier of the aggregated relationship `teaches` in Fig. 1.

Similarly for entities, an FD $Id(R) \rightarrow Y$ holds on R if Y is a proper attribute of R.

In addition, given a relationship R and two participants E_1 and E_2 of R, an FD $Id(E_1) \rightarrow Id(E_2)$ can then be inferred if the maximal participation of E_1 in R is 1. In our example, two FDs `Emp#` $\rightarrow$ `Dep#` on relations `Works` and `Heads` are deduced.

Moreover, for every participant E_i which is an aggregated relationship over $A_1,\ldots,A_n$, then propagate to R those FDs E_i: $Id(A_i) \rightarrow Id(A_j)$ provided that $Id(A_i)$ and $Id(A_j)$ are also attributes of R. For instance, suppose the cardinality between `Course` and `teaches` is (1,1) instead of (1,n); then an FD `Course#` $\rightarrow$ `Emp#` should be generated on `Teaches` and propagated to `Attends`.

As mentioned above, the identifier of a relationship as defined above $Id(R)$ may not be minimal due to FDs and thus some attributes may still be removed. If the cardinality of `Course` in `teaches` is (1,1) instead of (1,n), the identifier of `teaches` is `Course#` (instead of `{Course#, Emp#}` and the identifier of `Attends` is `{Stud#, Course#}` (instead of `{Stud#, Course#, Emp#}`). In the algorithms presented in Ref. [20], such a case is not taken into account and thus the generated relational schemas are less optimized.

Recursive relationships: This step is the same as the previous except that role names are introduced to avoid ambiguity. Our practical solution consists in concatenating role and identifier names in the relation mapping R. In the running example, relation `Supervises` is obtained with attributes `Supervisee_Emp#` and `Supervisor_Emp#`.

The FD `Supervisee_Emp#` $\rightarrow$ `Supervisor_Emp#` holds on the relation.

Aggregations: Each aggregation relationship A is mapped into a non normalized relation, comprising all single valued attributes of A and the identifiers of the composite class and the component class participating directly in A. In fact, the mapping rules for aggregation are those used for mapping simple relationships between two entities. Similarly for relationship mapping, FDs are generated from cardinalities.

Multivalued attributes: For each multivalued attribute A of an entity or a relationship E, create a new relation M that includes the identifier $Id(E)$ of E and A. In the running example, a relation `DepLocation` represents the multivalued attribute `Location` of entity `Department`.

An IND $M[Id(E)] \subseteq R[Id(E)]$ holds on relations M and R, mapping, respectively, a multivalued attribute A and an entity E; the inverse IND also holds if A is mandatory. In our example, the IND `DepLocation[Dep#]` $\subseteq$ `Department[Dep#]` and its inverse `Department[Dep#]` $\subseteq$ `DepLocation[Dep#]` are obtained.

INDs for relationships: Given a (aggregated) relationship $R$and a participant E of R, an IND $R[Id(E)] \subseteq E[Id(E)]$ can be inferred, where $Id(E)$ is recursively defined as shown previously. In our example, `Works[Dep#]` $\subseteq$ `Department[Dep#]` (`Department` is an entity) and `Attends[Emp#,Course#]` $\subseteq$ `Teaches[Emp#, Course#]` (`teaches` is an aggregated relationship) are deduced.

The inverse IND $E[Id(E)] \subseteq R[Id(E)]$ can also be deduced if the minimal participation of E in R is greater than 0. Hence, we can also deduce from Fig. 1 the IND `Department[Dep#]` $\subseteq$ `Works[Dep#]`.

INDs for recursive relationships: This step is the same as the previous one except that role names are taken into account, as for recursive relationships. For a concrete example, see Fig. 2 involving three INDs on relation `Supervises`.

INDs for aggregation: As mentioned above, since each part-whole relationship composing an aggregation can be considered in the mapping process as a simple relationship between two entities, INDs for aggregation will be generated following the same rules as INDs for simple binary relationships.

Derived relationships: Each derived relationship V is mapped into a relation R comprising only the attributes composing the identifiers of the participants of V since V possesses no proper attributes. For example, the derived relationship `DepartProf` is mapped into a view having the same name in Fig. 2.

An FD $X \rightarrow Y$ holds on R if it belongs to the closure of all FDs valid on any relation representing a component of V, provided that X and Y are also attributes of R. In our example, we can add to `DepartProf` the FD `EMP#` $\rightarrow$ `Dep#`, which is valid in relation `Works`.

Subset relationships: For two relationships R and S such that R is a subset of S, an IND $R[X] \subseteq S[X]$ is generated from R and S where X is the set of common attributes from R and S. In our example, the following IND is added `Heads[Dep#,Emp#]` $\subseteq$ `DepartProf[Dep#,Emp#]`.

Further, all FDs valid on S are also attached to R. Therefore, the FD `Emp#` $\rightarrow$ `Dep#`, is also attached to `Heads`.

Minimal covers: Construct a minimal cover for the FDs attached to each relation.

5.2. Relation normalization

The algorithm for decomposition into 3NF [6] consists of the following steps:

- make sure that the set of FDs is minimal;
- partition the set of FDs into groups such that all FDs in each group have equivalent left-hand sides;
- construct a relation for each group of FDs; and
- generate keys from left-hand sides of FDs.

In Fig. 2, relation `Employee` is normalized in two 3NF relations:

1. `Employee_a(Emp#,EmpName,Tel)` with FDs `Emp#` $\rightarrow$ `Tel` and `Emp#` $\rightarrow$ `EmpName`;
2. `Employee_b (Office, Tel)` with the FD `Office` $\rightarrow$ `Tel`.

Further in this step, the following relation keys are added: `Course#` on `Course`, `Stud#` on `Student`, `{Course#, Emp#}` on `Teaches`, `{Stud#,Emp#,Course#}` on `Attends`, `Emp#` on `Employee_a`, `Office` on `Employee_b`, `Emp#` on `Professor`, `Dep#` on `Department`, `Emp#` on `Heads`, `Emp#` on `DepartProf`, `{Dep#,Location}` on `DepLocation`, `Emp#` on `ProfMajor` and `Supervises_Emp#` on `Supervises`.

5.3. Database normalization

Specialize INDs: If relation R has been decomposed into 3NF relations R_i, then replace each IND of the form $R[X] \subseteq S[Y]$ (respectively $Q[Y] \subseteq R[X]$) by a set of INDs of the form $R_i[X'] \subseteq S[Y']$ (respectively $Q[Y'] \subseteq R_i[X']$), where X' is the intersection of X and the set of attributes of R_i, and Y' the subset of Y corresponding to X'. In our example, `Employee_a` replaces `Employee` in the following INDs: `Works[Emp#]` $\subseteq$ `Employee[Emp#]`, `Employee[Emp#]` $\subseteq$ `Works[Emp#]` and `Professor[Emp#]` $\subseteq$ `Employee[Emp#]`.

Eliminate redundant attributes: For each original or decomposed relation R, eliminate the superfluous attributes as well as the FDs and INDs involving these attributes by using the algorithm presented in Section 5.4. In our example, attribute `Dep#` in relation `Heads`, `Heads: Emp#` $\rightarrow$ `Dep#`, and `Head[Dep#]` $\subseteq$ `DepartProf[Dep#]` are removed.

Add INDs: For each non normalized relation R decomposed into a set of 3NF relations $R_1, \ldots, R_n$, add to the database all INDs of the form $R_i[X] \subseteq R_j[X]$, where X is the set of attributes common to R_i *and* R_j. In our example, the inclusion dependencies `Employee_a[Tel]` $\subseteq$ `Employee_b[Tel]` and `Employee_b[Tel]` $\subseteq$ `Employee_a[Tel]` are added.

Eliminate redundant relations: An all-key 3NF relation R is redundant with respect to another relation S if the INDs $R[U] \subseteq S[X]$ and $S[X] \subseteq R[U]$ hold, where U comprises all the attributes of R. Then, every relation R redundant with respect to a relation S must be eliminated, as well as the two INDs relating R and S. Further, attach all FDs of R to S and replace R by Sin all INDs having R in its left- or right-hand side. In our example, `Teaches` is redundant with respect to `Attends`. Thus, the two INDs relating `Teaches` and `Attends` are removed; `Attends` replaces `Teaches` in the three INDs.

5.4. Detecting superfluous attributes in a relation

For each attribute A of a relation R perform the following four steps.

Initialization: Construct the set K of keys K_i of R. If K only consists of a key containing all attributes of R (i.e. R is all-key), then no attribute A of R is superfluous. Otherwise, construct K', the set of keys of R not including A, temporarily remove all FDs involving A in R and all INDs involving a view V in its right-hand side, where attribute A of R is necessary to perform a join in the construction of V.

In Fig. 2, for relation `Heads` and the attribute

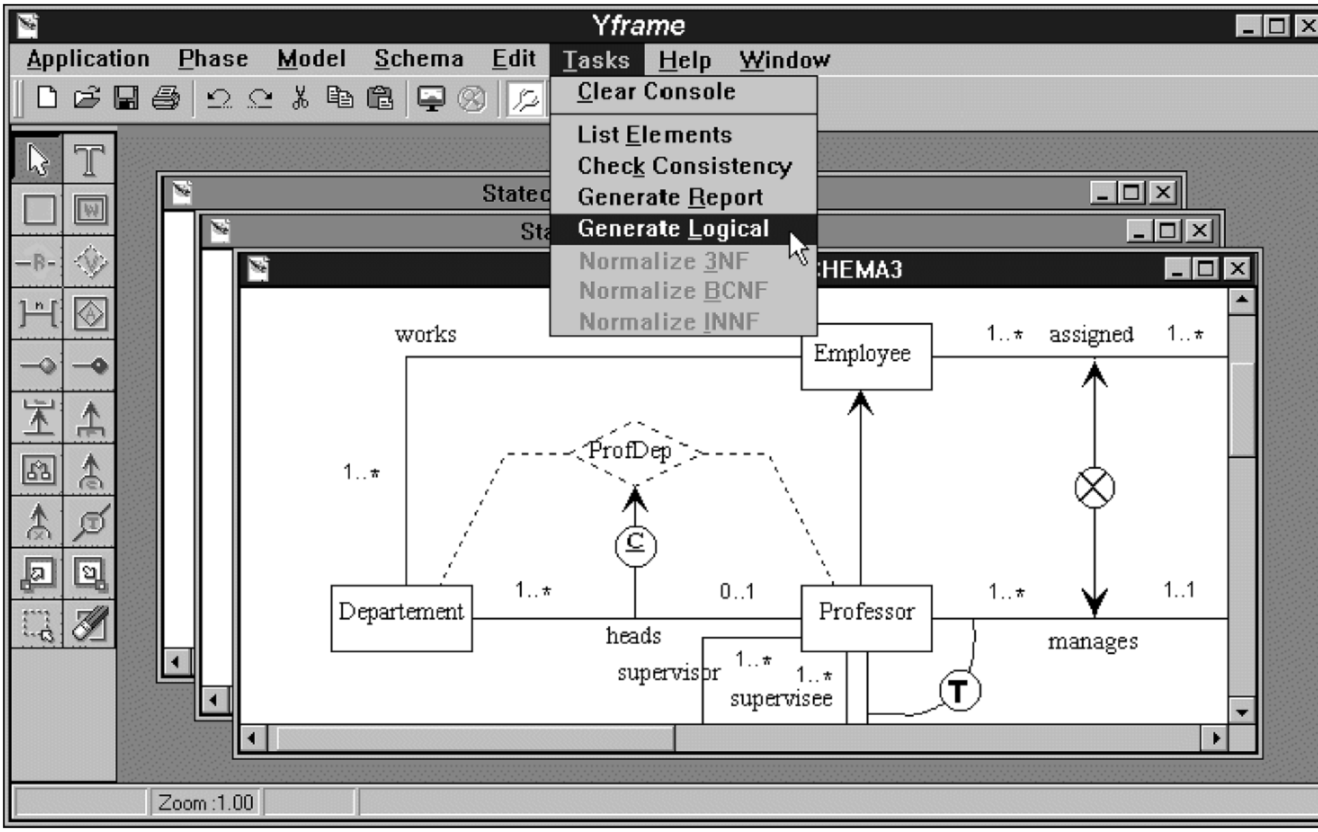

Fig. 8. Layout of **Y** *frame*.

Dep#,$K = K' = \{[Emp\#]\}$ and the FD Emp# → Dep# is temporarily removed.

Restorability test: If K' is not empty then choose any key K_i from K. If $K_i \rightarrow A$ cannot be deduced from the dependencies valid on the database (those that are not temporarily deleted), then A is not superfluous, otherwise A is restorable. In our example, the FD Emp# → Dep# can be deduced from the dependencies valid on the database. Thus Dep# is restorable.

Non essentiality test: If $K - K'$ is empty, then A (found to be restorable in the previous step) is superfluous. Otherwise, if there exists a key K_i of R containing A such that $K_i \rightarrow U$, where U is the set comprising all attributes of R, then A is superfluous. Otherwise, let C be the closure of K_i, and reinsert the dependencies temporarily removed. If $(C \cap U) - \{B\} \rightarrow U$ cannot be deduced then A is not superfluous. Otherwise, A is superfluous and insert into K' any key of R contained in $(C \cap U) - \{B\}$. In our example, since $K - K'$ is empty Dep# is non essential and thus superfluous.

Reinsert or specialize dependencies: If the attribute is not superfluous then reinsert the dependencies temporarily removed in the first step. Otherwise, add the INDs that can be deduced by transitivity using attribute A of R. Given the INDs $Q[X] \subseteq R[Y_1]$ and $R[Y_2] \subseteq S[Z]$, if $Y = Y_1 \cap Y_2$ and if $A \in Y$, then add the IND $Q[X'] \subseteq S[Z']$, where X' and Z' are the attributes corresponding to Y.

Then, replace every IND of the form $R[X] \subseteq S[Y]$ or $S[Y] \subseteq R[X]$ where $A \in X$ by $R[X'] \subseteq S[Y']$, and $S[Y'] \subseteq R[X']$ where $X' = X - \{A\}$ and Y' is the set of attributes corresponding to X' provided that X' is not empty.

6. Database design method in Prolog

The use of Prolog [13] for building our logical schema generator is essential to our approach. The declarative nature of Prolog gives it several advantages (clarity, modularity, conciseness, and legibility) over conventional programming languages and makes it more suitable for CASE prototype design. It also makes possible an integration between static knowledge of the world, or facts, and deductive statements, or rules.

6.1. ER+ representation in Prolog

We now present how ER+ concepts are represented by specific Prolog predicates.

ER+ conceptual schemas can be drawn directly on the screen of **Y** *frame* [29], our CASE shell (see Section 8), using a Graphical User Interface as shown in Fig. 8. The ER+ schema is validated using the integrity constraints defining the syntax and semantics of the ER+ abstractions.

This graphical representation is encoded by a set of Prolog predicates, which are introduced as the base of facts. These predicates form the input to the logical schema

```
entity(depart).
attribute(depart,depNo,1,1).
attribute(depart,depName,1,1).
attribute(depart,location,1,n).
identifier(depart,[depNo]).
entity(professor).
attribute(professor,status,1,1).
attribute(professor,major,1,n).

entity(course).
attribute(course,courseNo,1,1).
attribute(course,title,1,1).
attribute(course,nbHours,1,1).
identifier(course,[courseNo]).
entity(employee).
attribute(employee,empNo,1,1).
attribute(employee,tel,1,1).
identifier(employee,[empNo]).

entity(student).
attribute(student,studNo,1,1).
attribute(student,studName,1,1).
attribute(student,year,1,1).
attribute(student,year,1,1).
identifier(student,[studNo]).
attribute(employee,empName,1,1).
attribute(employee,office,1,1).
er_fd(employee,[office],[tel]).

relationship(teaches).
participates(teaches,professor,0,n,'').
participates(teaches,course,1,n,'').
relationship(works).
participates(works,employee,1,1,'').
participates(works,depart,1,n,'').
relationship(heads).
participates(heads,depart,1,n,'').
participates(supervises,professor,0,1,supervisor).
generalization(employee,partial,exclusive,'','').
derived_relationship(departProf,[professor]).

relationship(attends).
participates(attends,student,1,n,'').
participates(attends,teaches,1,n,'').
relationship(departProf).
participates(departProf,professor,1,1,'').
participates(departProf,depart,1,n,'').
participates(heads,professor,0,1,'').
relationship(supervises).
participates(supervises,professor,0,n,supervisee).
isa(professor,employee,'','').
subset_of(heads,departProf).
```

Fig. 9. Prolog facts representing the schema of Fig. 1.

generator, which automatically generates a relational schema normalized in IN-NF.

An entity `Ent` is represented by a predicate `entity(Ent)`.

A relationship `Rel` is represented by a predicate `relationship(Rel)`. Each participation of an entity or relationship `Part` into a relationship `Rel` is represented by a predicate `participates(Rel,Part,MinCard,MaxCard,Role)`, where `MinCard` and `MaxCard` are the minimal and the maximal cardinalities, and `Role` is the role of `Part` in `Rel` if `Part` participates in `Rel` several times.

An attribute `Attr` attached to an entity or relationship `Own` is represented by a predicate `attribute(Own,Attr,MinCard,MaxCard)`, where `MinCard` and `MaxCard` are the minimal and the maximal cardinalities.

An identifier of an entity `Ent` is represented by a predicate `identifier(Ent,Ident)`, where Ident is the list of attributes composing the identifier. As usual in Prolog, a list is represented by an expression between brackets, and members of the list are separated by commas.

A weak entity `Ent` is represented by a predicate `weak_entity(Ent,IdentRel)`, where `IdentRel` is the relationship relating `Ent` to one of its identifying owners. Entity `Ent` and relationship `IdentRel` are defined as above. Each weak entity `Ent` must have a (partial) `identifier` defined as above or define a predicate `identif_rels(Ent,RelLst)`, where `RelLst` is the list of identifying relationships that allow to uniquely identify instances of `Ent`. For example, in Fig. 6, `identif_rels(DailySeg,[TS,DTDS])` states that instances of `DailySeg` are identified by a combination of the identifiers of `Segment` and `DailyTrip`.

Two different predicates are used to represent generalizations. First, predicate `generalization(Super,TotalPart,ExclOver,Criteria,DefAttr)` defines a total/partial and exclusive/overlapping generalization of an entity `Super` according to a `Criteria`. Second, predicate `isa(Subcl,Supercl,Criteria,AttrValue)` states that `Subcl` is a subclass of `Supercl` according to a `Criteria`.

Notice that `Criteria` allows to represent parallel generalizations. An example is when an entity `employee` is specialized into `admin` and `tech` according to job type, and is also specialized into `hourly_paid` and `monthly_paid` according to salary.

`DefAttr` and `AttrValue` allow to represent predicate-defined generalizations. For example, `person` could be specialized into `child` and `adult` according to attribute `age`.

A derived relationship `DervRel` is represented, in addition to predicates `relationship` and `participates` as above, by a predicate `derived_relationship(DervRel,RelLst)`, where `RelLst` is the list of relationships and generalizations defining (through conjunction) the derived relationship.

A subset relationship `SubRel` is represented by a predicate `subset_of(Subcl,Supercl)`, where `Subcl` is a subclass of `Supercl`.

Explicit FDs attached to an entity or relationship `Own` are represented by a predicate `er_fd(Own,Left,Right)`, where `Left` and `Right` are the list of attributes composing each side of the FD.

Fig. 9 gives the encoding of the example ER+ schema of Fig.1.

6.2. Relational schema representation in Prolog

Relational facts are generated when the ER+ schema is mapped into a relational schema, during the first step. This relational schema is then modified during the second step for 3NF normalization, and during the third step for IN-NF normalization.

A relation `Rel` is represented by a predicate

```
rel_attrs(depart,[depNo,depName,location]).             rel_attrs(works,[empNo,depNo]).
rel_attrs(employee,[empNo,tel,empName,office]).         rel_attrs(heads,[empNo,depNo]).
rel_attrs(student,[studNo,studName,year]).              rel_attrs(departProf,[empNo,depNo]).
rel_attrs(attends,[empNo,courseNo,studNo]).             rel_attrs(professor,[empNo,major]).
rel_attrs(course,[courseNo,title,nbHours]).             rel_attrs(teaches,[empNo,courseNo]).
rel_attrs(supervisor,[supervisor_empNo,supervisee_empNo]).
fd(employee,[empNo],[empName]).                         fd(depart,[depNo],[depName]).
fd(employee,[empNo],[tel]).                             fd(depart,[depNo],[location]).
fd(employee,[empNo],[office]).                          fd(student,[studNo],[studName]).
fd(employee,[office],[tel]).                            fd(student,[studNo],[year]).
fd(course,[courseNo],[title]).                          fd(course,[courseNo],[nbHours]).
fd(professor,[empNo],[major]).                          fd(works,[empNo],[depNo]).
fd(departProf,[empNo],[depNo]).                         fd(heads,[empNo],[depNo]).
ind(attends,[studNo],student,[studNo]).                 ind(student,[studNo],attends,[studNo]).
ind(teaches,[courseNo],course,[courseNo]).              ind(works,[depNo],depart,[depNo]).
ind(course,[courseNo],teaches,[courseNo]).              ind(depart,[depNo],works,[depNo]).
ind(works,[empNo],employee,[empNo]).                    ind(departProf,[depNo],depart,[depNo]).
ind(employee,[empNo],works,[empNo]).                    ind(departProf,[empNo],professor,[empNo]).
ind(teaches,[empNo],professor,[empNo]).                 ind(professor,[empNo],employee,[empNo]).
ind(heads,[empNo,depNo],departProf,[empNo,depNo]).      ind(heads,[empNo],professor,[empNo]).
ind(attends,[courseNo,empNo],teaches,[courseNo,empNo]).  ind(heads,[depNo],depart,[depNo]).
ind(supervises,[supervisor_empNo],professor,[empNo]).    ind(depart,[depNo],heads,[depNo]).
ind(teaches,[courseNo,empNo],attends,[courseNo,empNo]).  view(departProf,[professor,works]).
```

Fig. 10. ER + -to-relational mapping: results of the first step.

rel_attrs(Rel,RelAttrs), where RelAttrs is the list of its attributes.

A key of a relation Rel is represented by the predicate key(Rel,AttrLst), where AttrLst is the list of attributes of the key.

A FD that holds on a relation Rel is represented by a predicate fd(Rel,Left,Right), where Left and Right are the list of attributes composing each side of the FD. Working with minimal covers requires the right-hand sides to be singletons. This is easily achieved by decomposing FDs.

An IND that holds on relations SubRel and SuperRel is represented by the predicate ind(SubRel,SubAttrs, SuperRel,SuperAttrs), where SubAttr s (respectively SuperAttr s) is the ordered list of attributes of SubRel (respectively of SuperRel) concerned by the IND. Moreover, SubAttrs and SuperAttrs are ordered in the same way.

Other predicates generated by the system are as follows: rel_all_key(Rel) represents an all-key relation Rel, tnfdecomp(Rel,Decomp) denotes that Decomp is a 3NF decomposition of relation Rel, superf(Rel, Attr) denotes a superfluous attribute Attr in a relation Rel, tnf(Rel) denotes that Rel is a 3NF relation, and innf(Rel) denotes a relation Rel with no superfluous attribute.

6.3. ER+-to-relational mapping rules

We give in Fig. 10 the result of applying the first step of the ER+-to-relational mapping to our example. This figure corresponds to the relational schema of Fig. 2.

6.4. Relational normalization

The second step of our method normalizes each 3NF relation and generates keys. For our example, the predicates given in Fig. 11 are added. Notice that relation employee is decomposed into employee_a and employee_b.

6.5. Database schema normalization

Database schema normalization algorithms described in

```
tnfdecomp(employee,employee_a). rel_attrs(employee_a,[empNo,tel,empName]).
tnfdecomp(employee,employee_b).        rel_attrs(employee_b,[tel,office]).
fd(employee_a,[empNo],[empName]).           fd(employee_a,[empNo],[tel]).
fd(employee_a,[empNo],[office]).           fd(employee_b,[office],[tel]).
key(depart,[depNo]).   key(employee_a,[empNo]).  key(employee_b,[office]).
key(student,[studNo]). key(course,[courseNo]).     key(professor,[empNo]).
key(works,[empNo]).    key(heads,[empNo]).          key(departProf,[empNo]).
key(depart,[depNo]).   key(employee_a,[empNo]).  key(employee_b,[office]).
key(student,[studNo]). key(course,[courseNo]).     key(professor,[empNo]).
key(works,[empNo]).    key(heads,[empNo]).          key(departProf,[empNo]).
key(teaches,[courseNo,empNo]).      key(attends,[studNo,empNo,courseNo]).
key(supervises,[supervisee_empNo]).
```

Fig. 11. 3NF relational normalization: results of the second step.

this subsection produce the final base of facts representing a database schema in IN-NF. Referring to our example, depNo is detected to be superfluous in Heads, and thus rel_attrs(Heads,[empNo,depNo]) is replaced by rel_attrs(Heads,[empNo]). Further, the following predicates are removed.

```
fd(Heads,[empNo],[depNo]).
ind(Heads,[depNo],depart,[depNo]).
ind(depart,[depNo],Heads,[depNo]).
```

Also, since relation teaches is redundant with respect to relation attends, the following predicates are removed:

```
rel_attrs(teaches,[empNo,courseNo]).
key(teaches,[empNo,courseNo]).
ind(attends,[empNo,courseNo],teaches,
[empNo,courseNo]).
ind(teaches,[empNo,courseNo],attends,
[empNo,courseNo]).
```

and the following predicates

```
ind(teaches,[empNo],professor,[empNo]).
ind(teaches,[courseNo],course,[courseNo]).
ind(course,[courseNo],teaches,[courseNo]).
```

are replaced by

```
ind(attends,[empNo],professor,[empNo]).
ind(attends,[courseNo],course,[courseNo]).
ind(course,[courseNo],attends,[courseNo]).
```

7. Application to a bus company

In this section, we apply our method to the information system of a bus company. The case has been adopted from Ref. [3] and we refer to Ref. [27] for further detail about the application.

Fig. 12 shows the ER+ schema depicting the system. Seat reservations are made directly by clients (passengers or travel agencies). Clients hold reservations or travel on a specific daily trip, actually composed of specific daily route segments. Trips can be ordinary or special and are composed of route segments. Buses and drivers are assigned to daily trips. The system keeps individual data on each driver and each bus including, respectively, absences and mechanical problems.

Fig. 13 shows the relational database schema obtained by applying classical ER+-to-relational mapping and normalization algorithms like those discussed in Section 5, in particular like those in Ref. [21,33]. To simplify the figure, we do not include FDs and INDs. Our method produces the same relational output but without the following relations trip_dailytrip, rtseg_dailyrtseg, trip_rtseg, dailytrip_dailyrtseg, driver_drvabs as indicated in Fig. 13.

Again, we refer to Ref. [27] for a complete resolution of the case study. These relations have been found redundant with respect to IN-NF normalization and then removed to generate the final database schema.

8. CASE shell context and related works

This work has been carried out in a larger project in which a prototype CASE shell for object-oriented information systems (and databases) development is being constructed [29,46,47]. The system called **Y** frame, is developed in LPA Prolog, and generates C++ code and relational database schemes in SQL for Oracle. It integrates concepts and models (ER+, Statecharts, Data Flow Diagrams, Object Interaction Graphs, Uses Cases) from different object-oriented methods (e.g. [7,8,15,18,39,45]). Of course, the relational database design module implements the method described here.

In an OO perspective, an application is described by several complementary models capturing static, dynamic, and functional aspects. Our CASE shell is characterized by a high degree of flexibility and a modular architecture: different abstractions and modeling mechanisms can be incorporated, customized and combined in each model. Similar models can be incorporated and combined by any method, thus allowing to customize the conceptual languages used to describe the system throughout the development lifecycle.

Since the early 1990s, much research has focused on CASE tool and/or expert system for database design [35] implemented in declarative langages, as these languages have advantages over imperative languages in a prototype development-environment. Although they are user-driven especially due to their declarative implementation, most of these research tools such as *View Creation System* [41], IBMS (Information Base Modeling System) [22] or DDEW (Database Design and Evaluation Workbench) [38] typically support only one or sometimes very few development methods specific to the tool. This kind of system, respectively, requires and provides inputs and outputs in one or a few implemented data models. Users and designers of such a tool, thus have a limited choice of design methods.

The IBMS system can be viewed as a good and recent example of these one-design-method tools. It can provide multiple functionalities and is compatible with other OO modeling perspectives, but the system has its own unique design method using the TSER (Two Stage Entity Relationship) approach [23]. The system consists of three classical components: (1) the modeling construct and interface facilities; (2) mapping algorithms; and (3) a design knowlegde base. The modeling construct is comprised of a semantic entity-relationship model for system analysis tasks and an operational entity-relationship model for database design. Following the TSER method, the mapping algorithms

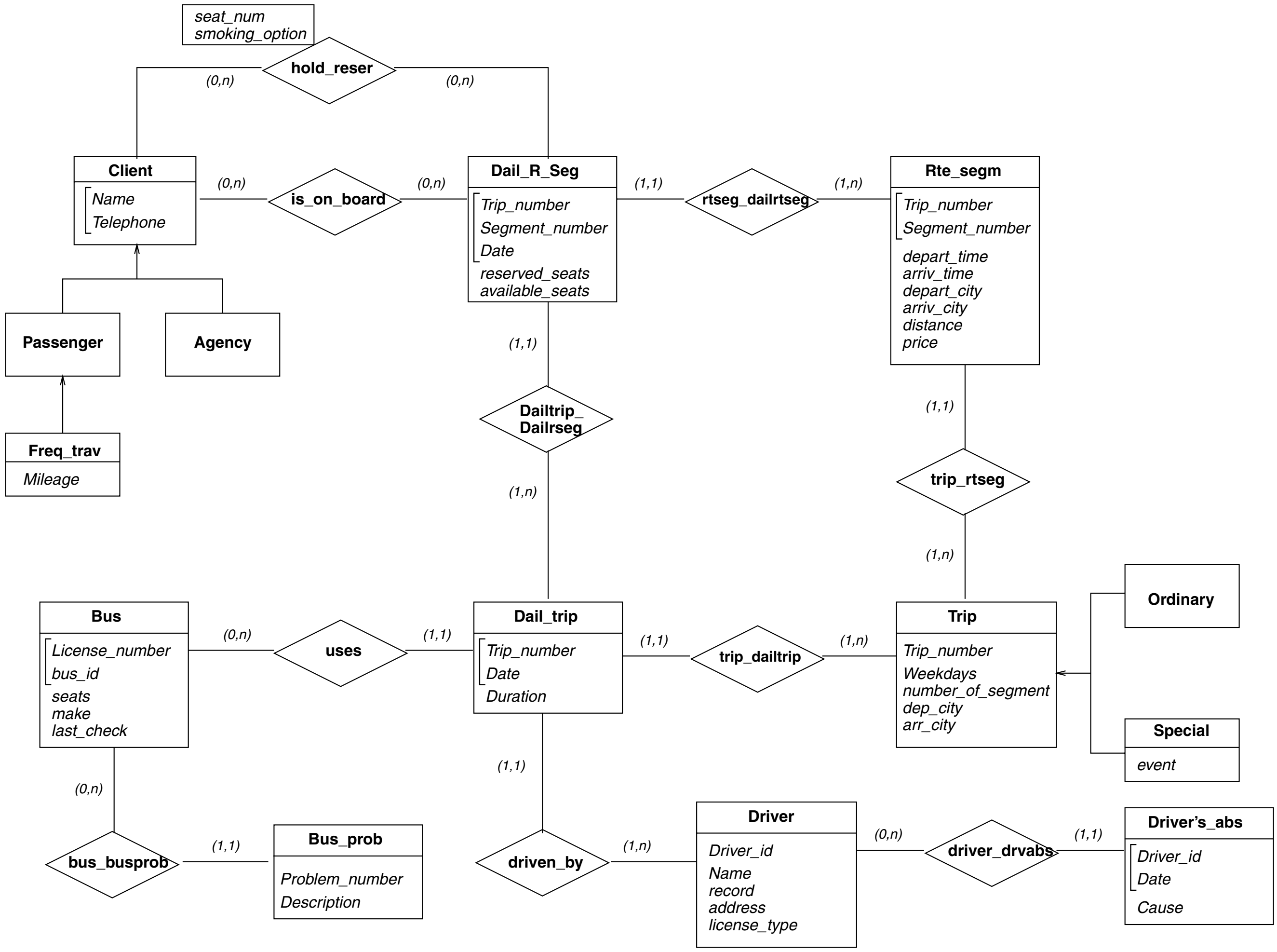

Fig. 12. A bus company.

integrate these two models and create normalized relational schemas and integrity rules. Finally, generic modeling rules and applications specific knowledge constitute the knowledge base that can be used both internally (by the mapping algorithms) and externally (by the user).

A few CASE tools use a less-limited design approach and integrate several methods customizable by the designer. Generally, they are called *CASE shell* rather than CASE tool.

Our tool, **Y** *frame*, is based on such a CASE shell approach. As mentioned, design methods are implemented and can use existing static, dynamic and functional models easily customizable by final users. Our relational database design module described in this paper can be compared to the design method of DDEW (Database Design and Evaluation Workbench) [38]. DDEW is a well-known research CASE tool, supporting database design from requirement specifications to final physical schema. It uses an extended ER model for the conceptual phase and provides the user with a choice of relational, network or hierarchical model for the logical step. As in our translation process, logical formulas and relational constraints as keys, functional, inclusion dependencies and sources of redundant relations are also taken into account in the ER+ schema, in addition to classical ER concepts mapping. Unlike our method, some other relational design issues like null values, view integration, design heuristics introduced from users are implemented. However, some conceptual enhancements like generalization or subset relationship, weak entities, aggregated, recursive and *n*-ary relationships are not supported. Again, database schema normalization is investigated as *global normalization* through the Universal Relation Assumption [5] and functional dependencies, but not formalized as inclusion normal form through inclusion dependencies and functional dependencies taken together.

9. Conclusions and further work

The main goal of our work has been to develop and test a method for relational database design based on Ref. [20], especially with respect to its viability in the context of CASE shell development.

We demonstrated the usefulness of the inclusion normal

Holds_reser

Name	Telephone	Trip_number	Segment_number	Date	Seat_number	Smoking_option

Rtseg_dailyrtseg

Trip_number	~~Segment_number~~	Date

Client

Name	Telephone

Is_on_board

Name	Telephone	Date	Trip_number	Segment_number

Daily_r_seg

Trip_number	Segment_number	Date	Reserved_Seats	Available_seats

Agency

Name	Telephone

Rte_Segm

Trip_number	Segment_number	Depart_city	Depart_time	Arriv_city	Arriv_time	Price	Distance

Passenger

Name	Telephone

Dailytrip_dailyrtseg

Trip_number	~~Segment_number~~	Date

Trip_dailytrip

Date	~~Trip_number~~

Trip_rtseg

Trip_number	~~Segment_number~~

Frequent_trav

Name	Telephone	Mileage

Daily_trip

Trip_number	Date	Duration

Trip

Trip_number	Dep_city	Arr_city	Weekdays	Number_of_Segments

Uses

Trip_number	Date	License_number

Driven_by

Trip_number	Date	Driver_id

Ordinary

Trip_number

Special

Trip_number	Event

Bus

License_number	Bus_id	Seats	Make	Last_check

Driver

Driver_id	Name	Address	License_type	Record

Driver_drvabs

~~Driver_id~~	~~Date~~

Bus_busprob

Problem_number	License_number

Bus_prob

Problem_number	Description

Drivers_abs

Driver_id	Date	Cause

Fig. 13. The bus company relational database.

form (IN-NF). Since ER cycles and inclusion constraints are often present in conceptual schemas, IN-NF normalization is needed to safely translate ER schemas into relational schemas.

Then, we presented our method for relational database design. It improves the algorithm of Ref. [20], in the following respects:

- takes into account multivalued attributes, weak entities and recursive relationships;
- distinguishes generalization and subset relationships, particularly being able to represent parallel generalizations,
- takes into account several identifiers for entities, particularly due to the inheritance of identifiers for (multiple) generalization,
- generates implicit FDs during the ER+-to-relational mapping,
- uses minimal covers instead of original sets of FDs,
- projects the INDs for 3NF decompositions,
- generates INDs that can be deduced by transitivity, before removing superfluous attributes, and specializes INDs after removing each superfluous attribute.

Several databases have been tested and have provided convincing results. We refer to Refs. [24,26,27,47] for further detail. We briefly described one of these applications in Section 7. These tests have allowed us to improve the mapping rules and the IN-NF normalization algorithms.

Another important result of our work is the development of an environment supporting relational database design. It helps the early detection of errors in the development lifecycle, which constitutes a necessity in software engineering. Our system validates the ER+ specifications introduced by the user, by performing integrity checking based on the syntax and semantics of the ER+ abstractions. Whenever errors are detected, the user is informed with appropriate explanations. Then our system allows the automatic generation of the corresponding relational database schema, normalized in IN-NF.

Several issues need to be further investigated. Concerning the ER+ formalism, abstractions like part-relationship [25] or materialization [16] could also be implemented. Part relationship is the link relating composites (e.g. car) to components (e.g. body and engine), while materialization describes the relationship between a class of object categories (e.g. models of cars) and a class of more concrete objects (e.g. individual cars).

Our mapping rules produce relations that keep track of the distinction between entities and relationships. Other more optimized rules can generate fewer relations but lose this semantic classification. Our system can be used to compare the pros and cons of each method. Optimization of our mapping rules may be, for instance, realized by implementing a relation merging algorithm [32].

Normalization algorithms implemented in our system only deal with FDs and INDs. The method could be enhanced by taking into account less common data dependencies like multivalued and join dependencies, take fourth and fifth normal forms into consideration, and thus shed light on their user-oriented semantics. We could also analyze the consequences of normalization into Boyce–Codd normal form (BCNF). This might be achieved with the algorithm of Tsou and Fisher [43]. Every relation of a database that is in IN-NF is only guaranteed to be in 3NF. However, as is well known, it is sometimes impossible to reach BCNF for a 3NF relation without losing dependency preservation.

References

[1] W. Armstrong, Dependency structures of database relationships, Proceedings of the IFIP Congress, Geneva, Switzerland, 1974, pp. 580–583.

[2] P. Atzeni, E. Chan, Independent database schemes under functional and inclusion dependencies, in: P. Stocker, W. Kent, P. Hammersley (Eds.), Proceedings of the 13th International Conference on Very Data Bases, VLDB '87, Brighton, England, Morgan Kaufmann, Los Altos, CA, 1987, pp. 159–166.

[3] C. Batini, C. Di Battista, Second generation data dictionaries: structure, design and usage, in: Tutorial at the EDBT'92 Conference, 1992.

[4] C. Batini, S. Ceri, S. Navathe, Conceptual Database Design: an Entity-Relationship Approach, Benjamin/Cummings, Menlo Park, CA, 1992.

[5] C. Beeri, P. Bernstein, N. Goodman, in: S. Yao (Ed.), Proceedings of the 4th International Conference on Very Large Data Bases, VLDB'78, West Berlin, Germany, Morgan Kaufmann, Los Altos, CA, 1978, pp. 113–124.

[6] P. Bernstein, Synthesising third normal form relations from functional dependencies, ACM Transactions on Database Systems 1 (4) (1976) 277–298.

[7] G. Booch, Object-oriented Analysis and Design with Applications, 2nd ed., Benjamin/Cummings, Menlo Park, CA, 1994.

[8] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language: User Guide, Addison-Wesley, Reading, MA, 1999.

[9] M. Bouzeghoub, G. Gardarin, E. Metais, Database design tools: an expert system approach, in: A. Pirotte, Y. Vassiliou (Eds.), Proceedings of the 11th International Conference on Very Large Data Bases, VLDB '85, Stockholm, Sweden, Morgan Kaufmann, Los Altos, CA, 1985, pp. 82–95.

[10] M. Casanova, R. Fagin, C. Papadimitriou, Inclusion dependencies and their interaction with functional dependencies, Journal of Computer and System Sciences 28 (1) (1984) 29–54.

[11] M. Casanova, L. Tucherman, A. Furtado, A. Braga, Optimization of relational schemas containing inclusion dependencies, in: P. Apers, G. Wiederhold (Eds.), Proceedings of the 15th International Conference on Very Large Data Bases, VLDB '89, Amsterdam, The Netherlands, Morgan Kaufmann, Los Altos, California, 1989, pp. 317–325.

[12] S. Ceri, G. Gottlob, Normalization of relations and Prolog, Communications of the ACM 29 (6) (1986) 524–544.

[13] W. Clocksin, C. Mellish, Programming in Prolog, Springer, Berlin, Germany, 1984.

[14] E. Codd, Further normalization of the database relational model, Data Base Systems 6 (1972) 33–64.

[15] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, P. Jeremaes, Object-oriented development: the fusion method, Prentice Hall, Englewood Cliffs, NJ, 1994.

[16] M. Dahchour, A. Pirotte, E. Zimányi, Metaclass implementation of materialization. Technical report YEROOS TR-99/01, IAG-QANT,

Université catholique de Louvain, Belgium, February 1999, (submitted for publication).

[17] A. Doğaç, B. Yuruten, S. Spaccapietra, A generalized expert system for database design, IEEE Transactions on Software Engineering 15 (4) (1989) 479–491.

[18] D. D'Souza, A. Wills, Objects, components, and frameworks with UML: the catalysis approach, Addison-Wesley, Reading, Massachusetts, 1998.

[19] R. Elmasri, S. Navathe, Fundamentals of database systems, 2nd ed., Benjamin/Cummings, Menlo Park, CA, 1994.

[20] C. Goh, Towards a Viable Methodology for Logical Relational Database Design. Masters thesis, National University of Singapore, 1992.

[21] F. Golshani (Ed.), Proceedings of the 8th International Conference on Data Engineering, ICDE'92, Tempe, Arizona IEEE Computer Society, 1992.

[22] C. Hsu, Two-stage Entity-relationship (TSER): Concepts Guide, Rensselaer Polytechnic Institute, 1997.

[23] C. Hsu, A. Perry, M. Bouziane, W. Cheung, TSER: a data modeling system using the two-stage entity-relationship approach, in: S. March (Ed.), Proceedings of the 6th International Conference on the Entity-Relationship Approach, ER '87, New York, USA, 1987, pp. 580–583.

[24] M. Kolp, Elaboration d'un système expert pour la conception de bases de données relationnelles à partir du modèle entité-association. Mémoire de Licence Spéciale en Sciences de l'Information et de la Documentation, INFODOC, Université Libre de Bruxelles, September 1994.

[25] M. Kolp, A. Pirotte, An aggregation model and its C++ implementation, in: M. Orlowska, R. Zicari (Eds.), Proceedings of the 4th International Conference on Object-Oriented Information Systems, OOIS'97, Brisbane, Australia, November 1997, pp. 211–224.

[26] M. Kolp, E. Zimányi, Prolog-based algorithms for database design, Proceedings of the 6th International Conference on Practical Applications of Prolog, PAP'98, London, UK, March 1998.

[27] M. Kolp, E. Zimányi, Enhanced ER to relational mapping and its Prolog implementation for CASE shell. Technical report YEROOS TR-95/01, IAG-QANT, Université catholique de Louvain, Belgium, April 1995.

[28] M. Kolp, E. Zimányi, Relational database design using an ER approach and Prolog, in: S. Bhalla (Ed.), Proceedings of the 6th International Conference on Information Systems and Management of Data, CISMOD'95, LNCS 1006, Bombay, India, Springer, Berlin, Germany, 1995, pp. 214–231.

[29] M. Kolp, E. Zimányi, **Y** *frame*: a modular and declarative framework for software engineering. Technical report YEROOS TR-00/04, IAG-QANT, Université catholique de Louvain, Belgium, March 2000.

[30] T. Ling, C. Goh, Logical database design with inclusion dependencies, in: F. Golshani (Ed.), Proceedings of the 8th International Conference on Data Engineering, ICDE'92, Tempe, Arizona, 1992, pp. 642–649.

[31] T. Ling, F. Tompa, T. Kameda, An improved third normal form for relational databases, ACM Transactions on Database Systems 6 (2) (1981) 329–346.

[32] M. Markowitz, Merging relations in relational databases, in: F. Golshani (Ed.), Proceedings of the 8th International Conference on Data Engineering, ICDE'92, Tempe, Arizona, 1992, pp. 428–437.

[33] M. Markovitz, A. Shoshani, Representing extended entity-relationship structures in relational databases: a modular approach, ACM Transactions on Database Systems 17 (3) (1992) 423–464.

[34] J. Mitchell, Inferences rules for functional and inclusion dependencies, Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS '83, 1983, pp. 58–69.

[35] H.A. Muller, R.J. Norman, J. Slonim, Computer Aided Software Engineering, Kluwer Academic, Dordrecht, 1996.

[36] W. Potter, multi-model schema design tool in Prolog, Proceedings of the 1st International Conference on Expert Database Systems, 1984, pp. 747–759.

[37] A. Rochfeld, P. Negros, Relationship of relationship and other interrelationship links in ER models, Data & Knowledge Engineering 9 (1992) 205–221.

[38] A. Rosenthal, D. Reiner, Tools and transformations -rigorous and otherwise- for practical database design, ACM Transactions on Database Systems 19 (2) (1994) 167–211.

[39] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[40] V. Storey, A selective survey of the use of artificial intelligence for database design systems, Data & Knowledge Engineering 11 (1993) 61–102.

[41] V. Storey, R. Goldstein, Design and development of an expert database design system, International Journal of Expert Systems 3 (1) (1990) 31–63.

[42] T. Teorey, Database Modeling and Design: the Entity-relationship Approach, Morgan Kaufmann, Los Altos, CA, 1990.

[43] D. Tsou, P. Fischer, Decomposition of a relation scheme into Boyce–Codd normal form, ACM-SIGACT 14 (3) (1982) 23–29.

[44] J. Ullman, Principles of Database Systems, Computer Science Press, Rockville, Maryland, 1982.

[45] E. Yu, Modelling Strategic Relationships for Process Reengineering, PhD thesis, University of Toronto, Department of Computer Science, 1995.

[46] E. Zimányi, M. Kolp, Using Prolog to implement a CASE shell for object-oriented development, Proceedings of the 8th Symposium and Exhibition on Industrial Application of Prolog, INAP'95, Tokyo, Japan, October 1995, pp. 41–48.

[47] E. Zimányi, M. Kolp, A Prolog-based architecture for an object-oriented CASE shell, in: Proceedings of the 4th International Conference on Practical Application of Prolog, PAP'96, London, UK, April 1996, pp. 497–520.