Prolog-Based Algorithms for Database Design *

Manuel Kolp † Esteban Zimányi ‡

Abstract

This paper develops a method that maps an enhanced Entity-Relationship (ER+) schema into a relational schema and normalizes the latter into inclusion normal form (IN-NF). Unlike classical normalization that characterizes individual relations only, IN-NF takes inter-relational redundancies into account. We describe a Prolog implementation of the method, developped in the context of a CASE shell for software development.

Keywords: Inclusion Normal Form, ER+ model, relational model, database design, Prolog, CASE.

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 decomposition of database design into four steps [3, 12, 28]: 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 [4]. 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 were defined [14, 17, 18] to integrate the interaction of constraints in the database and detect redundancies.

Nowadays, relational database design 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 [12, 14, 19, 26] for ER+ to relational translations but the semantic distance between the two models can lead to anomalies in the logical schema.

*This work is part of the YEROOS (Yet another project on Evaluation and Research on Object-Oriented Strategies) project, principally based at the University of Louvain. See http://yeroos.qant.ucl.ac.be

†University of Louvain, IAG-QANT, 1 Place des Doyens, 1348 Louvain-La-Neuve, Belgium, e-mail: kolp@qant.ucl.ac.be.

‡University of Brussels, INFODOC, 50 Av. F.D. Roosevelt, CP 175-2, 1050 Brussels, Belgium, e-mail: ezimanyi@ulb.ac.be

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. We improved the ER+-to-relational mapping and the database normalization rules given in [14] to take into consideration enhanced ER+ mechanisms [16].

Since database design is complex, a significant research development has been the adoption of knowledge-based techniques for automating design [25]. In the context of Computer-Aided Software Engineering (CASE) shell design, we implemented in Prolog the algorithms constructing a normalized relational schema from an ER+ one enhancing the proposals of [6, 9, 11, 23].

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 normalization theory and introduces the new normal forms. Our enhancement of the design method based on [14] is explained in Section 4. The implementation of the three steps of the method, ER+-to-Relational mapping, relational normalization, and database normalization are explained in Sections 5 to 7. Our experiences of using Prolog for developing the CASE shell are discussed in Section 8. Finally, Section 9 gives conclusions and points to 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 Figure 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. Figure 1 shows a recursive relationship: supervises where Professor plays the roles of supervisor and supervisee.

Cardinality constraints model restrictions to relationships. In Figure 1, every instance of Department participates in at least 1 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 EmpName. 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 omitted from figures. In Figure 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 exactly identify one instance of the entity. In Figure 1, Emp# identifies Employee.

Several abstraction mechanisms were 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, Figure 5 (b) [3] 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 Figure 5 (b), 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 which, when combined, uniquely identify weak entity instances. In Figure 5 (b), instances of

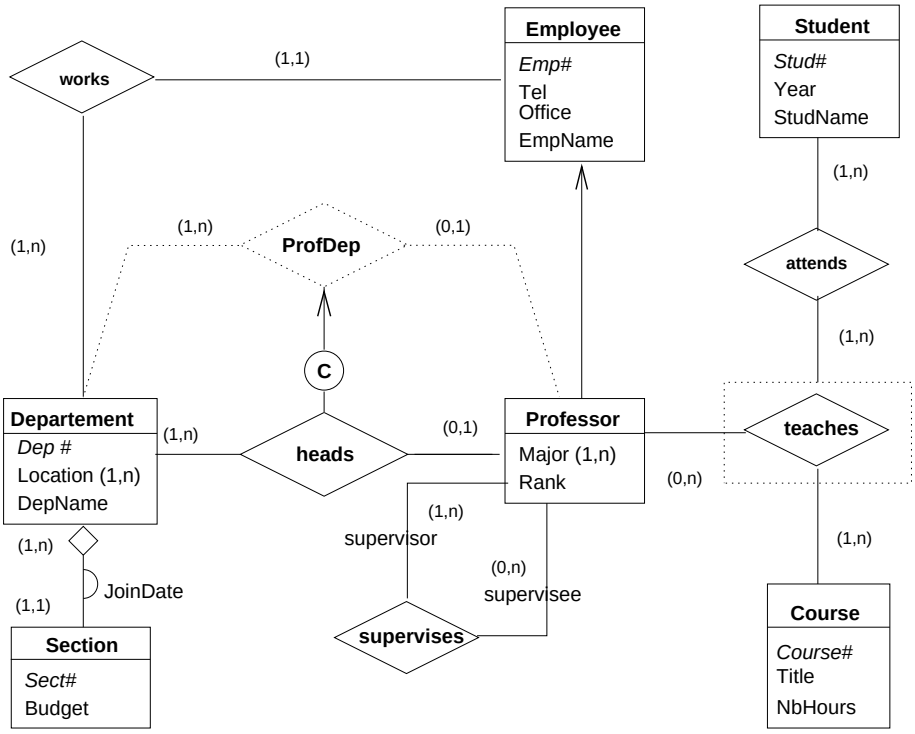

Figure 1: An ER+ schema

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(ies)*. A superentity may have several subentities and vice versa. A superentity may also be a subentity of another superentity. A subentity inherits all attributes and relationships of its superentities. Figure 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 Figure 1, the composite entity Department is composed of Sections. As for generalization, composite and component entities may participate in other aggregations, leading to aggregation hierarchies. Similarly to relationships, cardinalities model restrictions between composites and components in aggregation hierarchies. Also aggregations can have attributes, as shown by attribute joinDate.

An *aggregated relationship* models a relationship as a *participant* in another relationship. For instance, the aggregated relationship teaches in Figure 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* [14, 17, 24] 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 Figure 1, DepartProf is a derived relationship.

A *subset constraint* models an inclusion 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. Unlike [17], we differentiate generalization and subset constraints, since they correspond to different abstraction mechanisms.

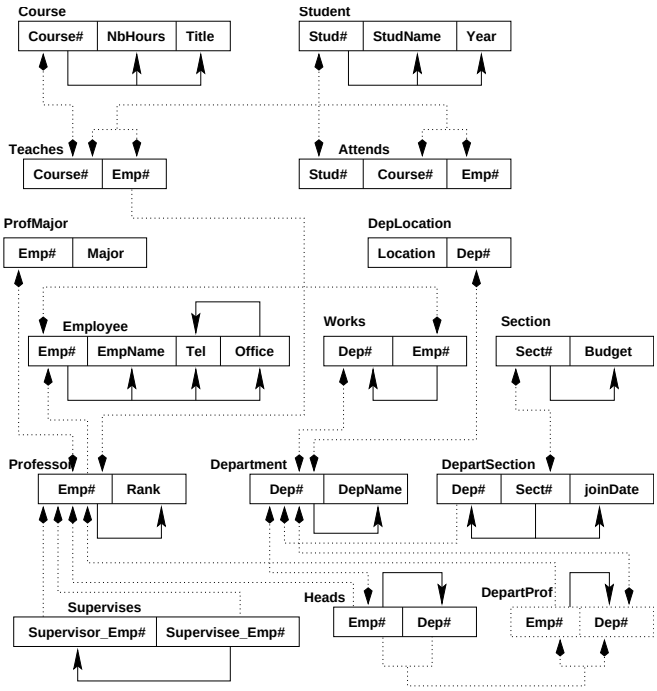

Figure 2: A set of nonnormalized relations from the ER+ schema of Figure 1.

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

Figure 2 shows a relational database schema obtained by applying an ER+-to-relational mapping to the ER+ schema of Figure 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 allows to detect these redundancies.

Data dependencies are constraints on databases and relations [13, 28]. This paper only deals with functional and inclusion dependencies (FDs and INDs).

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

INDs are interrelational constraints on pairs of relations represented, as in Figure 2, by dashed arrows: the IND Attends[Stud#] ⊆ 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 [7] 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 [7, 21]:

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 and 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 Figure 1, Stud# $\rightarrow$ StudName holds on entity Student; this constraint also holds on the corresponding relation in Figure 2. In the same way, FDs explicitly represented in an ER+ schema also hold in the corresponding relations.

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 Figure 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 gives the mapping rules deducing all implicit FDs and INDs in an ER+ schema and attach them to the corresponding relational schema.

3 Database Normalization

Normalization [4, 10, 28] was introduced in relational database design to avoid redundancies and 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.

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, 8, 17, 18].

To circumvent these problems, in [18] is introduced the *Improved third normal form* (Improved 3NF). Unlike classical normal forms, Improved 3NF considers several relations rather than individual relations and determines redundancies with respect to FDs. Normalization into Improved 3NF consists in detecting and deleting superfluous attributes. It was proven that if a database is in Improved 3NF, then each individual relation is in 3NF [18].

Inclusion normal form (IN-NF) [14, 17] was later introduced to guarantee databases without redundancies with respect to FDs *and* INDs. It was also proven that if a database is in IN-NF, it is also in Improved 3NF [14, 17].

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: if an attribute appears in two or more places in a database schema, then it refers to the same notion, it represents the same semantics.

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

ER cycles and its associated inclusion constraints become possible sources of superfluous attributes in the corresponding relational schemas. In cycles, some information can be deduced in more than one way: in Figure 1, for a Professor who heads a Departement, the Department in which he works can be deduced either via heads or via works. Moreover, some information is not necessary to deduce other information: in Figure 1, a Department can be headed by more than one Professor. Consequently, there is no constraint holding on heads by which a particular Department determines one and only one Professor.

While in the ER+ schema heads captures some important real-world semantics (a Professor can head a Department), in the relational schema of Figure 2, Dep# in Heads is said

to be *restorable* since Heads : Emp# $\rightarrow$ Dep# can be deduced from Heads[Dep#, Emp#] $\subseteq$ DepartProf[Dep#, Emp#] and from DepartProf : Emp# $\rightarrow$ Dep#. It is also said to be *nonessential* since it is not needed to deduce other information, i.e., it is not part of the left-hand side of any FD holding on Heads. Therefore, Dep# is said to be *superfluous* and should be removed from relation Heads. Note also that all dependencies involving Dep# in Heads must be removed.

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.

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 $K \rightarrow A$ can be inferred from $\Sigma_{R(A)}$.

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

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

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

The main difference between Improved 3NF and IN-NF is that, for inferring an FD $X \rightarrow Y$, Improved 3NF considers only FDs while IN-NF considers both FDs and INDs[1].

We consider now redundant relations. In Figure 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 Figure 2 should also be deleted because it is redundant with respect to ProfMajor. Both ProfMajor[Emp#] $\subseteq$ Professor[Emp#] and Professor[Emp#] $\subseteq$ ProfMajor[Emp#] hold meaning that all information contained in Professor is also represented in ProfMajor.

Notice also that all attributes of Teaches are included in relation Attends. Since both 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 not present in Attends, such as semester, then it would not be redundant.

4 Design Method

Traditionally, database design has been accomplished using normalization. However, since the adoption of conceptual models in the mid-70's, normalization theory ceased to be the main

[1]For this reason, what we call restorable (nonessential, superfluous) attributes are called weakly restorable (weakly nonessential, weakly superfluous) attributes in [14, 17].

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 3, ER cycles can be sources of superfluous attributes not detected by classical normalization. Hence, the interest of enhanced ER-based design methods removing anomalies due to cycles and inclusion constraints.

We propose an integrated design method including normalization into IN-NF based on [Goh92]. These algorithms comprise three main steps:

(1) ER+-to-Relational Mapping (see also [12, 14, 19, 26, 28]): an ER+ schema is mapped into a set of nonnormalized 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 Bernstein algorithm [5]. Since several papers present in detail this algorithm and its implementation (see, e.g., [9]), we do not develop this phase in the paper.
(3) Database normalization: superfluous attributes and relations are deleted producing a database in IN-NF.

Sections 5 to 7 describe the Prolog predicates implementing these steps.

4.1 ER+ representation in Prolog

We shown next how ER+ concepts are represented by specific Prolog predicates.

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

A relationship Rel is represented by a predicate relationship(Rel). An aggregation Aggr is represented by a predicate aggregation(Aggr). Each participation of an entity or relationship Part into a relationship or aggregation 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 of an entity, relationship, or aggregation 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.

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. 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 allowing to uniquely identify instances of Ent.

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(Subent,Superent, Criteria,AttrValue) states that Subent is a subentity of Superent 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 hourlyPaid and monthlyPaid 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 constraint is represented by a predicate subset_of(Subent,Superent) where Subent is a subentity of Superent.

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

Figure 3 gives the encoding of the example ER+ schema of Figure 1.

```
entity(depart).                    entity(course).                    entity(student).
attribute(depart,depNo,1,1).       attribute(course,courseNo,1,1).    attribute(student,studNo,1,1).
attribute(depart,depName,1,1).     attribute(course,title,1,1).       attribute(student,studName,1,1).
attribute(depart,location,1,n).    attribute(course,nbHours,1,1).     attribute(student,year,1,1).
identifier(depart,[depNo]).        identifier(course,[courseNo]).     attribute(student,year,1,1).
entity(professor).                 entity(employee).                  identifier(student,[studNo]).
attribute(professor,status,1,1).   attribute(employee,empNo,1,1).     attribute(employee,empName,1,1).
attribute(professor,major,1,n).    attribute(employee,tel,1,1).       attribute(employee,office,1,1).
                                   identifier(employee,[empNo]).      er_fd(employee,[office],[tel]).
relationship(teaches).                                relationship(attends).
participates(teaches,professor,0,n,'').               participates(attends,student,1,n,'').
participates(teaches,course,1,n,'').                  participates(attends,teaches,1,n,'').
relationship(works).                                  relationship(departProf).
participates(works,employee,1,1,'').                  participates(departProf,professor,1,1,'').
participates(works,depart,1,n,'').                    participates(departProf,depart,1,n,'').
relationship(heads).                                  participates(heads,professor,0,1,'').
participates(heads,depart,1,n,'').                    relationship(supervises).
participates(supervises,professor,0,1,supervisor).   participates(supervises,professor,0,n,supervisee).
generalization(employee,partial,exclusive,'','').     isa(professor,employee,'','').
derived_relationship(departProf,[professor]).         subset_of(heads,departProf).
aggregation(departSection).                           participates(departSection,depart,0,n,'').
participates(departSection,section,1,1,'').           attribute(departSection,joinDate,1,1).
```

Figure 3: Prolog facts representing the schema of Figure 1.

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

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

A FD holding on a relation Rel is represented by a predicate fd(Rel,Left,Right) where Left and Right are lists 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 holding on relations SubRel and SuperRel is represented by predicate ind(SubRel, SubAttrs,SuperRel,SuperAttrs) where SubAttrs (resp. SuperAttrs) is the ordered list of attributes of SubRel (resp. 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.

5 ER+ to Relational Mapping

We show in this section the Prolog predicates implementing the ER+-to-relational mapping. Figure 4 shows the result of applying this first step to our example. This figure corresponds to the relational schema of Figure 2.

```
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(departSect,[depNo,sectNo,joinDate]).       rel_attrs(section,[secNo,budget]).
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]).
fd(departSect,[sectNo],[depNo]).                     fd(departSect,[sectNo],[joinDate]).
fd(section,[sectNo],[budget]).                       ind(departSection,[sectNo],section,[sectNo]).
ind(section,[sectNo],departSection,[sectNo]).        ind(departSection,[deptNo],depart,[deptNo]).
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]). ind(depart,[depNo],departProf[depNo]).
view(departProf,[professor,works]).
```

Figure 4: ER+ to relational mapping: results of the first step.

Generating Identifiers In the relational model, the information about a particular entity and the relationships in which it participates is scattered along several tables related with foreign keys. Thus, before realizing the ER+-to-relational mapping, one of the identifiers of each entity is chosen for serving as key in the relational model. Also, identifiers must be inherited from superentities to subentities, as well as inherited for strong entities to weak entities. We describe next the predicates implementing this preliminary step of the mapping algorithm.

```
gen_ident :- entity(Ent),\+(weak_entity(Ent,_)),identifier(Ent,Ident),
  length(Ident,Len),forall(identifier(Ent,Ident1),(length(Ident1,Len1),Len1>=Len)),
  \+(gen_identifier(Ent,_)),assert(gen_identifier(Ent,Ident)),fail.
gen_ident.
```

Predicate gen_ident chooses for each (regular) entity Ent one among the identifiers having the least number of attributes. This identifier is used in the relational schema to represent the links relating Ent to other entities or relationships.

```
inher_ident :- isa(Ent,SuperEnt,_,_),inher_ident(Ent,SuperEnt),fail.
inher_ident.

inher_ident(Ent,SuperEnt) :- gen_identifier(Ent,Ident),gen_identifier(SuperEnt,Ident),!.
inher_ident(Ent,SuperEnt) :- ident_ent(SuperEnt,Ident),\+(gen_identifier(Ent,Ident)),
  assert(gen_identifier(Ent,Ident)),add_attr(Ident,Ent).

ident_ent(Ent,Ident) :- gen_identifier(Ent,Ident),!.
ident_ent(Ent,Ident) :- isa(Ent,SuperEnt,_,_),ident_ent(SuperEnt,Ident),\+(gen_identifier(Ent,Ident)),
  assert(gen_identifier(Ent,Ident)),add_attr(Ident,Ent),!.
```

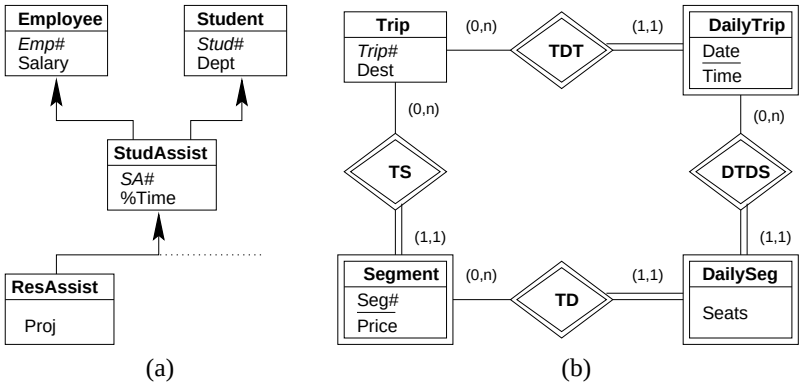

Figure 5: Examples of generalization hierarchies and weak entities

Predicate inher_ident attaches the identifier of a superentity to all its direct subentities. These inherited attributes are used as keys when joining the relations corresponding to superentities and to subentities. In the presence of multiple inheritance, an entity Ent inherits one identifier from each superentity. Further, Ent may have its own (locally defined) identifier. In any case, only one identifier is inherited from Ent to its immediate subentities.

Consider the example of Figure 5 (a). Since StudAssist is a direct subentity of both Employee and Student, Emp# and Stud# are inherited to StudAssist. Thus, StudAssist has three identifiers but only one of them is needed to be inherited to ResAssist.

Weak entities also inherit an identifier from each identifying entity. Consider the example shown in Figure 5 (b). Each weak entity either has 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, th e identifiers of DailyTrip and Segment are, respectively, {Trip#,Date} and {Trip#,Seg#}. For the second case, DailySeg can be identified by a combination of the identifiers of its identifying entities DailyTrip and Segment, i.e., the identifier is {Trip#,Seg#,Date}.

```
inher_ident_weak :- weak_entity(WeakEnt,_),\+(gen_identifier(WeakEnt,_)),
  ident_weak_ent(WeakEnt,Ident),fail.
inher_ident_weak.

ident_weak_ent(WeakEnt,Ident) :- gen_identifier(WeakEnt,Ident),!.
ident_weak_ent(WeakEnt,Ident) :- weak_entity(WeakEnt,IdentRel),identifier(WeakEnt,PartialIdent),
  participates(IdentRel,IdentEnt,_,_,_),IdentEnt\=WeakEnt,
  ( weak_entity(IdentEnt,_) -> ident_weak_ent(IdentEnt,SuperIdent)
  ; gen_identifier(IdentEnt,SuperIdent)
  ),append(SuperIdent,PartialIdent,Ident),
  assert(gen_identifier(WeakEnt,Ident)),add_attr(SuperIdent,WeakEnt),!.
ident_weak_ent(WeakEnt,Ident) :- identif_rels(WeakEnt,IdentRels),
  ident_weak_ent(IdentRels,WeakEnt,[],Ident),assert(gen_identifier(WeakEnt,Ident)),
  add_attr(Ident,WeakEnt).

ident_weak_ent([],WeakEnt,SoFar,Ident) :- list_set(SoFar,Ident).
ident_weak_ent([IdentRel}IdentRelLst],WeakEnt,SoFar,Ident) :-
  participates(IdentRel,IdentEnt,_,_,_),IdentEnt\=WeakEnt,ident_weak_ent(IdentEnt,SuperIdent),
  append(SoFar,SuperIdent,SoFar1),ident_weak_ent(IdentRelLst,WeakEnt,SoFar1,Ident).
```

The inheritance of indentifiers for weak entities is realized by predicate inher_ident_weak. The second clause of inher_ident_weak/2 is used when the weak entity has a partial identifier, while the third clause is used when instances of the weak entity are identified by a combination of several identifying relationships.

```
add_attr([],SubEnt).
add_attr([IdAttr}IdAttrLst],SubEnt) :-
  ( \+(attribute(SubEnt,IdAttr,1,1)) -> assert(gen_attribute(SubEnt,IdAttr,1,1))
  ; true
  ),add_attr(IdAttrLst,SubEnt).
```

When an entity inherits an identifier, all attributes composing the identifier must be attached to the entity. This is done by predicate add_attr. Predicate gen_attribute is used instead of attribute to differentiate the user-defined attributes from the attributes inherited during the mapping. Thus the original schema introduced by the user is not modified.

Mapping entities

```
map_entities :- entity(Ent),findall(Attr,attribute(Ent,Attr,_,1),AttrLst),
  findall(Attr,gen_attribute(Ent,Attr,_,1),GenAttrLst),append(AttrLst,GenAttrLst,AllAttrs),
  \+(rel_attrs(Ent,AllAttrs)),assert(rel_attrs(Ent,AllAttrs)),fail.
map_entities :- entity(Ent),
  ( gen_identifier(Ent,Ident) ; \+(weak_entity(Ent,_)),identifier(Ent,Ident) ),
  rel_attrs(Ent,AttrLst),member(Attr,AttrLst),\+(member(Attr,Ident)),
  \+(fd(Ent,Ident,[Attr])),assert(fd(Ent,Ident,[Attr])),fail.
map_entities.
```

Predicate map_entities maps each (regular or weak) entity Ent into a nonnormalized relation of the same name by adding a new predicate rel_attrs to the base of facts. The relation contains all mono-valued attributes of Ent, whether user-defined or inherited in the previous step. For instance, entity Student in Figure 1 is mapped into relation Student in Figure 2.

The second clause of the predicate generates all FDs implied by the identifiers. Each FD has, as left-hand side, an identifier of Ent and, as right-hand side, an attribute which is not part of that identifier. In the example, Student : Stud# $\rightarrow$ StudName is generated. Notice that identifiers inherited for generalization and weak entities (gen_identifier) generate also FDs.

```
gen_expl_fd :- er_fd(Class,Left,Right),member(Attr,Right),
  \+(fd(Class,Left,[Attr])),assert(fd(Class,Left,[Attr])),fail.
gen_expl_fd.
```

Predicate gen_expl_fd generates for every FD explicitly represented on the ER+ schema a set of equivalent FDs holding in the corresponding relation. Such FDs have only one element in the right-hand side. In our example, Employee : Office $\rightarrow$ Tel also holds on relation Employee.

```
gen_inds_gener :- generalization(Super,Crit,_,_,_),isa(Sub,Super,Crit,_),
  gen_identifier(Sub,Ident),gen_identifier(Super,Ident),\+(ind(Sub,Ident,Super,Ident)),
  assert(ind(Sub,Ident,Super,Ident)),fail.
gen_inds_gener.
```

Predicate gen_inds_gener asserts an IND between a relation SubEnt representing a subentity and a relation SuperEnt representing its nearest superentity. The attributes appearing in the IND are those comprising the common identifier of both entities. In our example, the IND Professor[Emp#] $\subseteq$ Employee[Emp#] is asserted.

Mapping Relationships

```
search_ident(Rel,PartLst,Ident) :- gen_identifier(Rel,Ident),!.
search_ident(Rel,PartLst,Ident) :- search_ident1(PartLst,[],Ident),assert(gen_identifier(Rel,Ident)).

search_ident1([],IdentLst,Ident) :- !,list_set(IdentLst,Ident).
search_ident1([[Ent,Role]}EntLst],IntLst,IdentLst) :- entity(Ent),gen_identifier(Ent,Ident),
  add_role(Role,Ident,NewIdent),append(IntLst,NewIdent,IntLst1),!,
  search_ident1(EntLst,IntLst1,IdentLst).
search_ident1([[Rel,Role]}EntLst],IntLst,IdentLst) :- relationship(Rel),
  part_relationship(Rel,PartLst),search_ident(Rel,PartLst,IdentPartLst),
  append(IntLst,IdentPartLst,IntLst1),!,search_ident1(EntLst,IntLst1,IdentLst).
```

Predicate search_ident calculates the identifier of a relationship, comprising the identifiers of all entities participating directly or indirectly in the relationship through aggregated relationships. The identifier of a relationship R relating a set of participants $E_1, \ldots, E_n$ 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;
- if Ei is an aggregated relationship over $A_1, \ldots, A_n$ then $Id(E_i) = Id(A_1) \cup \ldots \cup Id(A_n)$.

The second clause of search_ident1 deals with entities and concatenates role and entity identifier names to take recursive relationships into account (predicate add_role/3) while aggregated relationships are considered in the last clause. When all identifiers have been collected, repeated identifiers are removed with list_set, and a predicate gen_identifier is asserted for the relationship.

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

In the presence of FDs, the identifier of a relationship as defined above may not be minimal. 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 [14] such a case is not taken into account and thus the generated relational schemas are less optimized.

```
add_role('',Ident,Ident) :- !.
add_role(Role,Ident,NewIdent) :- add_role(Ident,Role,[],NewIdent) .
add_role([],Role,NewIdent,NewIdent) .
add_role([Attr}AttrLst],Role,SoFar,NewIdent) :- cat([Role,'_',Attr],NewAttr,_),
  append(SoFar,[NewAttr],SoFar1),add_role(AttrLst,Role,SoFar1,NewIdent) .
```

Role names are introduced for recursive relationships to avoid ambiguity. Predicate add_role generates new identifiers by adding Role as prefix to each attribute Attr included in an identifier Ident of an entity. In the example, the identifier of relation Supervises is Supervisee_Emp#.

```
map_relationships :- relationship(Rel),findall([Part,Role],participates(Rel,Part,_,_,Role),PartLst),
  assert(part_relationship(Rel,PartLst)),fail.
map_relationships :- relationship(Rel),part_relationship(Rel,PartLst),
  map_relationships(Rel,PartLst),find_impl_fd(Rel),fail.
map_relationships :- relationship(Rel),part_relationship(Rel,PartLst),search_fd(PartLst,Rel),fail.
map_relationships.
```

Predicate map_relationships maps each nonderived relationship Rel into a nonnormalized relation with the same name comprising all mono-valued attributes of Rel and the identifiers of Rel. The first clause computes in PartLst all entities or aggregated relationships participating in Rel and their roles. The second clause constructs the relation schemas corresponding to the relationships (predicate map_relationships/2) and generates the FDs corresponding to the participations with maximal cardinality of 1 (find_impl_fd). Finally, the third clause inherits the FDs from participants (which are aggregated relationships) to the relationships (search_fd). These predicates are explained next.

```
map_relationships(Rel,PartLst) :- derived_relationship(Rel,RelLst),assert(view(Rel,RelLst)),
  search_ident(Rel,PartLst,Ident),\+(rel_attrs(Rel,Ident)),assert(rel_attrs(Rel,Ident)),!.
map_relationships(Rel,PartLst) :- \+(derived_relationship(Rel,_)),search_ident(Rel,PartLst,Ident),
  findall(Attr,attribute(Rel,Attr,_,1),AttrLst),append(AttrLst,Ident,AttrLst1),
  \+(rel_attrs(Rel,AttrLst1)),assert(rel_attrs(Rel,AttrLst1)),add_fd_rel(AttrLst,Rel,Ident).
```

Predicate map_relationships/2 creates a relation schema for every relationship Rel. For derived relationships, the attributes of this schema are the identifiers of entities participating (directly

or indirectly) in Rel; also a predicate view is added to the fact base. For example, the derived relationship DepartProf is mapped into a view having the same name in Figure 2. For nonderived relationships, the schema has, in addition, the mono-valued attributes of the relationship.

```
add_fd_rel([],Rel,Ident).
add_fd_rel([Attr}AttrLst],Rel,Ident) :- \+(fd(Rel,Ident,[Attr])),
  assert(fd(Rel,Ident,[Attr])),add_fd_rel(AttrLst,Rel,Ident).
```

For a relationship Rel having attributes, predicate add_fd_rel generates FDs having as right-hand side the identifier of Rel and as left-hand side one of its attributes.

```
find_impl_fd(Rel) :- participates(Rel,Part,_,1,Role),\+(weak_entity(_,Rel)),
  one(gen_identifier(Part,Ident)),find_impl_fd(Rel,Part,Role,Ident),fail.
find_impl_fd(Rel).

find_impl_fd(Rel,Part,Role,Ident) :- participates(Rel,OtherPart,_,_,OtherRole),
  (Part \= OtherPart;Role \= OtherRole),one(gen_identifier(OtherPart,OtherIdent)),
  add_role(Role,Ident,NewIdent),add_role(OtherRole,OtherIdent,NewOtherIdent),
  member(Attr,NewOtherIdent),assert(fd(Rel,NewIdent,[Attr])),fail.
find_impl_fd(Rel,Part,Role,Ident).
```

Predicate find_impl_fd asserts implicit FDs holding on the relation representing a relationship Rel. The left-hand sides of these FDs are identifiers of entities participating in Rel with minimal cardinality equal to 1. The right-hand sides are the attributes composing the identifiers of the other entities participating in Rel. In our example, two FDs Emp# $\rightarrow$ Dep# on relations Works and Heads are deduced. If Rel is recursive, left-hand and right-hand sides are distinguished by the role names added as prefix to the attributes composing the identifiers.

```
search_fd([],Rel).
search_fd([Part}PartLst],Rel) :- entity(Part),!,search_fd(PartLst,Rel).
search_fd([Part}PartLst],Rel) :- relationship(Part),down_fd(Rel,Part),!,search_fd(PartLst,Rel).

down_fd(Rel,Part) :- rel_attrs(Rel,AttrLst),fd(Part,Left,Right),sublist_check(Left,AttrLst),
  sublist_check(Right,AttrLst),not(fd(Rel,Left,Right)),assert(fd(Rel,Left,Right)),fail.
down_fd(Rel,Part).
```

Consider a relationship Rel having at least one aggregated relationship Part as participant. Predicates search_fd and down_fd attach to relation Rel the FDs holding in relation Part provided that they concern only attributes appearing in Rel. 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.

```
make_temp_rel(Rel,RelLst,TempRel) :- search_derv_attr(RelLst,[],AttrLst),cat([Rel,'_v'],TempRel),
  \+(rel_attrs(TempRel,AttrLst)),assert(rel_attrs(TempRel,AttrLst)),assert(derv(TempRel,Rel)).
```

An FD $X \rightarrow Y$ holds on a derived relation R if it belongs to the closure of all FDs valid on any relation defining R provided that X and Y are also attributes of R. In our example, the FD Emp# $\rightarrow$ Dep# valid on relation Works can be added to DepartProf.

For determining the FDs valid on a derived relationship Rel, predicate make_temp_rel creates a temporary relation schema which is the join of all relations defining Rel. The name of the temporary relation is the name of Rel with the suffix _v.

```
search_derv_attr([],SoFar,ResLst) :- !,list_set(SoFar,ResLst).
search_derv_attr([Rel|RelLst],SoFar,ResLst) :- rel_attrs(Rel,AttrLst),
  append(SoFar,AttrLst,SoFar1),!,search_derv_attr(RelLst,SoFar1,ResLst).
```

Predicate search_derv_attr collects all attributes of relations defining a derived relationship.

```
gen_fd_derv_rel :- derived_relationship(DervRel,RelLst),make_temp_rel(DervRel,RelLst,TempRel),
  member(Rel,RelLst),fd(Rel,Left,Right),\+(fd(TempRel,Left,Right)),assert(fd(TempRel,Left,Right)),fail.
gen_fd_derv_rel :- retract(derv(TempRel,Rel)),project_fd(TempRel,Rel),
  retract(rel_attrs(TempRel,_)),retractall(fd(TempRel,_,_)),fail.
gen_fd_derv_rel.
```

Predicate gen_fd_derv_rel attachs to the temporary relation associated to a view DervRel all FDs holding on relations Rel composing the view. Then, it projects these FDs over the attributes of DervRel by calling project_fd.

```
gen_inds_rel :- participates(Rel,Part,MinCard,MaxCard,Role),one(gen_identifier(Part,Ident)),
  add_role(Role,Ident,NewIdent),assert(ind(Rel,NewIdent,Part,Ident)),
  ( MinCard \= 0 -> assert(ind(Part,Ident,Rel,NewIdent)) ; true ),fail.
gen_inds_rel.
```

Given a relationship Rel and an entity or aggregated relationship Part directly participating in Rel, predicate gen_inds_rel generates an IND between relation Rel and relation Part. The attributes appearing in the IND are those comprising the identifier of Part. 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.

If Part mandatorily participates in Rel (minimal cardinality is not equal to 0), the inverse IND is also generated. Hence, the IND Department[Dep#] $\subseteq$ Works[Dep#] can also be deduced.

Mapping Aggregations Each aggregation relationship A is mapped into a nonnormalized relation comprising all mono-valued attributes of A and the identifiers of the composite and component entities participating directly in A. In fact, the mapping rules for aggregation are those used for mapping simple relationships between two entities. Thus, FDs and INDs for aggregation are generated as for simple relationships.

Mapping Multivalued Attributes

```
map_multiv_attr :- attribute(Own,Attr,MinCard,MaxCard),( MaxCard=n ; MaxCard>1 ),
  cat([Own,'_',Attr],RelName),one(identifier(Own,Ident)),append(Ident,[Attr],AttrLst),
  assert(rel_attrs(RelName,AttrLst)),assert(ind(RelName,Ident,Own,Ident)),
  ( MinCard \= 0 -> assert(ind(Own,Ident,RelName,Ident)) ; true ),fail.
map_multiv_attr.
```

Predicate map_multiv_attr creates a relation for each multivalued attribute Attr of an entity or a relationship Own. The relation name is the concatenation of the entity (or relationship) name Own and the attribute name Attr. The relation has as attributes the attributes composing the identifier of Own as well as Attr. In the running example, a relation DepLocation with attributes Dep# and Location represents the multivalued attribute Location of Department.

Further map_multiv_attr adds an IND between the relation corresponding to the multivalued attribute and the relation corresponding to the entity (or relationship). If the attribute is mandatory (MinCard is not equal to 0) it also adds the inverse IND. In our example, the IND DepLocation[Dep#] $\subseteq$ Department[Dep#] and its inverse Department[Dep#] $\subseteq$ DepLocation[Dep#] are obtained.

Mapping Subset Constraints

```
gen_inds_subset :- subset_of(Rel1,Rel2),rel_attrs(Rel1,Rel1Attrs),
  rel_attrs(Rel2,Rel2Attrs),intersection(Rel1Attrs,Rel2Attrs,Inter),
  \+(ind(Rel1,Inter,Rel2,Inter)),assert(ind(Rel1,Inter,Rel2,Inter)).
gen_inds_subset.
```

Predicate gen_inds_subset asserts INDs corresponding to the subset constraints. If a relationship Rel1 is a subset of Rel2, an IND Rel1[Inter] $\subseteq$ Rel2[Inter] is generated where Inter is the set of common attributes from Rel1 and Rel2. In our example, the following IND is added Heads[Dep#, Emp#] $\subseteq$ DepartProf[Dep#, Emp#].

Inheriting FDs

```
inher_fd :- isa(Ent,_,_,_),inher_fd_ent(Ent),fail.
inher_fd :- find_cycles,subset_of(Rel,_),inher_fd_rel(Rel),fail.
inher_fd :- retractall(fd_generated(Super)),retractall(cycle(Cycle)).

inher_fd_ent(Ent) :- fd_generated(Ent),!.
inher_fd_ent(Ent) :- isa(Ent,SuperEnt,_,_),\+(fd_generated(SuperEnt)),inher_fd_ent(SuperEnt),fail.
inher_fd_ent(Ent) :- isa(Ent,SuperEnt,_,_),project_fd(SuperEnt,Ent),fail.
inher_fd_ent(Ent) :- \+(fd_generated(Ent)),assert(fd_generated(Ent)).
```

Predicate inher_fd inherits FDs between superentities and subentities, as well as between relationships defined as subsets of other relationships.

As shown in inher_fd_ent, this inheritance must be realized traversing the generalization hierarchy in breadth-first order, i.e., to be able to inherit the FDs from SuperEnt to Ent, SuperEnt must have already inherited all FDs from all its superentities. This is controlled by asserting predicates fd_generated.

Predicate inher_fd_rel (not shown) is similar to inher_fd_ent, but it takes into account cycles between relationships connected through subsets, thus defining equality between all relationships appearing in the cycle. In this case, when inheriting FDs, all relationships appearing in a cycle are treated as one equivalence class.

Generating Minimal Covers Construct a minimal cover for the FDs attached to each relation. Predicate project_fd, used for projecting a set of FDs over a set of attributes, is not explained here. The reader can find more details about this predicate in papers dealing with 3NF normalization of relation schemas in Prolog [9].

6 Relation Normalization

The second step of our method normalizes each 3NF relation and generates keys. The algorithm for decomposition into 3NF consists of the following steps [5]:

- 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.

We refer to, e.g., [9] for a complete implementation of 3NF normalization in Prolog.

After applying this step to our running example shown in Figure 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 relation keys are generated. Thus, the predicates given in Figure 6 are added to the fact base.

```
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]).
```

Figure 6: 3NF relational normalization: results of the second step.

7 Database Normalization

The normalization algorithms described in this section 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]).
```

Implementing Axioms for INDs

```
closure_fd_ind(Rel,AttrLst,Closure) :-
  ( fd(Rel,Left,Right)
  ; tnfdecomp(OrigRel,Rel),tnfdecomp(OrigRel,Rel1),Rel\=Rel1,fd(Rel1,Left,Right)
  ),sublist_check(Left,AttrLst),not sublist_check(Right,AttrLst),
  union(AttrLst,Right,NewAttrLst),!,closure_fd_ind(Rel,NewAttrLst,Closure).
closure_fd_ind(Rel,AttrLst,Closure) :- trans_ind(Rel,RelAttrs,Super,SuperAttrs),
  fd(Super,Left,[Right]),sublist_check(Left,SuperAttrs),member(Right,SuperAttrs),
  corresp_attr(SuperAttrs,RelAttrs,Left,SubLeft),corresp_attr(SuperAttrs,RelAttrs,[Right],[SubRight]),
  sublist_check(SubLeft,AttrLst),not(member(SubRight,AttrLst)),
  union(AttrLst,[SubRight],NewAttrLst),!,closure_fd_ind(Rel,NewAttrLst,Closure).
closure_fd_ind(Rel,Closure,Closure).
```

Predicate closure_fd_ind realizes the closure of a set of attributes AttrLst with respect to a set of FDs and INDs. The first clause concerns the generation of a new attribute of the closure using FDs; the second uses INDs. As can be seen in the first clause, one must take into account both FDs directly attached to relation Rel and FDs attached to a relation Rel1 such that Rel and Rel1 are both decompositions of the same relation. The second clause implements the Pullback rule, i.e., derives a new attribute of the closure using a (transitive) IND between relations Rel and Super, and an FD attached to relation Super.

```
trans_ind(Rel,RelAttrs,Super,SuperAttrs) :- trans_ind(Rel,RelAttrs,Super,SuperAttrs,INDsLst).

trans_ind(Rel,RelAttrs,Super,SuperAttrs,[ind(Rel,RelAttrs,Super,SuperAttrs)]) :-
    ind(Rel,RelAttrs,Super,SuperAttrs).
trans_ind(Rel,SubInter,Super,SuperInter,[ind(Rel,SubAttrs,Int,IntAttrs1)}SoFar]) :-
  ind(Rel,SubAttrs,Int,IntAttrs1),trans_ind(Int,IntAttrs2,Super,SuperAttrs,SoFar),
  ( member(ind(Rel,SubAttrs,Int,IntAttrs1),SoFar) -> !,fail
  ; intersection(IntAttrs1,IntAttrs2,Inter),Inter\=[],
  corresp_attr(IntAttrs2,SuperAttrs,Inter,SuperInter),corresp_attr(IntAttrs1,SubAttrs,Inter,SubInter)
  ).
```

Predicate trans_ind searches for INDs between relations Rel and Super. Such INDs can be either defined in the base or can be derived using transitivity. To avoid infinite loops when there are cyclic INDs, the list of INDs already used in the computation is kept in the fifth argument. This condition is tested with predicate member. Recall that cyclic INDs are very common in particular due to the INDs generated for entities participating mandatorily in relationships.

Specialize INDs

```
spec_ind_dec :- tnfdecomp(Rel,_),retract(ind(Rel,X,SuperRel,Y)),tnfdecomp(Rel,Dec),
  rel_attrs(Dec,DecAttrs),intersection(X,DecAttrs,X1),X1 \= [],corresp_attr(X,Y,X1,Y1),
  not(ind(Dec,X1,SuperRel,Y1)),assert(ind(Dec,X1,SuperRel,Y1)),fail.
spec_ind_dec :- tnfdecomp(Rel,_),retract(ind(SubRel,Y,Rel,X)),tnfdecomp(Rel,Dec),
  rel_attrs(Dec,DecAttrs),intersection(X,DecAttrs,X1),X1 \= [],corresp_attr(Y,X,Y1,X1),
  not(ind(SubRel,Y1,Dec,X1)),assert(ind(SubRel,Y1,Dec,X1)),fail.
spec_ind_dec.
```

If relation Rel has been decomposed into 3NF relations Dec, predicate spec_ind_dec replaces each IND of the form Rel[X] $\subseteq$ SuperRel[Y] (resp. SubRel[Y] $\subseteq$ Rel[X]) by a set of INDs of the form Dec[X1] $\subseteq$ SuperRel[Y1] (resp. SubRel[Y1] $\subseteq$ Dec[X1]) where X1 is the intersection of X and the set of attributes of Dec, and Y1 the subset of Y corresponding to X1. In our example, Employee_a replaces Employee in the three INDs Works[Emp#] $\subseteq$ employee[emp#], employee[emp#] $\subseteq$ Works[emp#], and professor[emp#] $\subseteq$ employee[emp#].

Eliminate superfluous 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 7.1. In our example, attribute Dep# in relation Heads, Heads : Emp# $\rightarrow$ Dep#, and Heads[Dep#] $\subseteq$ DepartProf[Dep#] are removed.

Add INDs

```
gen_ind_dec :- tnfdecomp(Rel,Dec1),tnfdecomp(Rel,Dec2),Dec1 \= Dec2,rel_attrs(Dec1,Attr1),
  rel_attrs(Dec2,Attr2),intersection(Attr1,Attr2,Inter),Inter \= [],not(ind(Dec1,Inter,Dec2,Inter)),
  assert(ind(Dec1,Inter,Dec2,Inter)),not(ind(Dec2,Inter,Dec1,Inter)),
  assert(ind(Dec2,Inter,Dec1,Inter)),fail.
gen_ind_dec.
```

Once all superfluous attributes are removed, the INDs for the decompositions can be generated. For each couple of 3NF relations Dec1 and Dec2 belonging to the decompositon of relation Rel, predicate gen_ind_dec asserts INDs of the form Dec1[Inter] $\subseteq$ Dec2[Inter] where Inter is the set of attributes common to Dec1 and Dec2. 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

```
rem_red_rel :- tnfdecomp(Rel,Rel1),tnfdecomp(Rel,Rel2),Rel1\=Rel2,
  rel_attrs(Rel1,Attr1),verify_all_key(Rel1,Attr1),rel_attrs(Rel2,Attr2),
  sublist_check(Attr1,Attr2),attach_deps(Rel1,Rel2),remove_relation(Rel1),fail.
rem_red_rel :- rel_attrs(Rel,RelAttrs),not(tnfdecomp(Rel,_)),verify_all_key(Rel,RelAttrs),
  rel_attrs(SuperRel,Attr2),Rel \= SuperRel,not(tnfdecomp(SuperRel,_)),not(view(SuperRel,_)),
  ind(Rel,RelAttrs,SuperRel,SubsetAttr2),ind(SuperRel,SubsetAttr2,Rel,RelAttrs),
  retract(ind(Rel,RelAttrs,SuperRel,CorrAttrs)),retract(ind(SuperRel,CorrAttrs,Rel,RelAttrs)),
  attach_deps(Rel,SuperRel),remove_relation(Rel),fail.
rem_red_rel.
```

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 attributes of R. The first clause of rem_red_rel remove redundant relations belonging to the decomposition of an original relation Rel. The second clause removes redundant relations in the general case.

In both cases, if R is redundant with respect to S, the two INDs must be eliminated. Further, all FDs of R must be attached to S and R must be replaced by S in all INDs having R in its left- or right-hand side. This is done with predicate attach_deps, shown next.

```
attach_deps(Rel1,Rel2) :- retract(fd(Rel1,Left,Right)),assert(fd(Rel2,Left,Right)),fail.
attach_deps(Rel1,Rel2) :- retract(ind(Rel1,Attrs,SuperRel,SuperAttrs)),
  assert(ind(Rel2,Attrs,SuperRel,SuperAttrs)),fail.
attach_deps(Rel1,Rel2) :- retract(ind(SubRel,SubAttr,Rel1,Attr)),
  assert(ind(SubRel,SubAttr,Rel2,Attr)),fail.
attach_deps(Rel1,Rel2).
```

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.

7.1 Detecting superfluous attributes in a relation

Checking superfluous attributes

```
rem_sup_attr(Rel) :- rel_attrs(Rel,RelAttrs),rem_sup_attr(RelAttrs,Rel).

rem_sup_attr([],Rel).
rem_sup_attr([Attr|RelAttrs],Rel) :- key_attr_set(Rel,KeyLst),
  part_keys_attr(KeyLst,Attr,KeysWith,KeysWithout),assert(superf(Rel,Attr)),
  rem_dep_temp(Rel,Attr),check_rest(KeysWithout,Rel,Attr),check_nonessent(KeysWith,Rel,Attr),
  write('Attribute '),write(Attr),write(' is superfluous in '),write(Rel),nl,
  rem_attr(Rel,Attr),!,rem_sup_attr(RelAttrs,Rel).
rem_sup_attr([Attr|RelAttrs],Rel) :- reins_dep(Rel),
  retract(superf(Rel,Attr)),!,rem_sup_attr(RelAttrs,Rel).
```

Predicate rem_sup_attr removes superfluous attributes in a relation Rel. It computes the set of keys in KeyLst and partition this set into the sets KeysWith and KeysWithout including or not the attribute Attr. Predicate rem_dep_temp temporarily removes the FDs containing attribute Attr as well as all INDs concerning views for which the attribute Attr is necessary to perform a join. In Figure 2, for relation Heads and attribute Dep#, KeyLst = KeyWithout = {[Emp#]}, KeyWith = {}, and the FD Emp# → Dep# is temporarily removed.

If the attribute is superfluous, the predicates check_rest and check_nonessent succeed, and then rem_attr removes the attribute. Then it tests if other superfluous attributes can be detected.

Otherwise, if the attribute is not superfluous, one of check_rest or check_nonessent fails and the third clause is taken, in which the dependencies temporarily removed are reinserted with reins_dep, and the test for other superfluous attributes continues.

```
rem_dep_temp(Rel,Attr) :- fd(Rel,Left,Right),( member(Attr,Left) -> true ; member(Attr,Right) ),
  retract(fd(Rel,Left,Right)),assert(remember_dep(fd(Rel,Left,Right))),fail.
rem_dep_temp(Rel,Attr) :- view(View,Expr),member(Rel,Expr),member(Rel1,Expr),Rel\=Rel1,
  rel_attrs(Rel,RelAttrs),rel_attrs(Rel1,RelAttrs1),intersection(RelAttrs,RelAttrs1,[Attr]),
  retract(ind(SubRel,SubAttrs,View,ViewAttrs)),
  assert(remember(ind(SubRel,SubAttrs,View,ViewAttrs))),fail.
rem_dep_temp(Rel,Attr).
```

Before testing if an attribute Attr of Rel is nonessential, predicate rem_dep_temp is used to remove temporarily FDs and INDs involving this attribute. The first clause removes FDs in which the attribute appears. The second clause removes all INDs of the form $R[X] \subseteq S[Y]$ where S is a view constructed from relations in the database in which the attribute Attr is necessary in some join.

Restorability Test

```
check_rest([],Rel,B) :- !.
check_rest(KeysWithout,Rel,Attr) :- member(Key,KeysWithout),
  ( closure_fd_ind(Rel,Key,Clos),member(Attr,Clos) -> true ; fail ).
```

As shown in check_rest, attribute Attr is restorable if either (1) all keys contain attribute Attr, and thus the first clause is taken, or (2) if there is a Key not containing the attribute such that Key $\rightarrow$ Attr can be deduced from the dependencies valid on the database (those that are not temporarily deleted). In our example, the FD Emp# $\rightarrow$ Dep# can be deduced from the dependencies valid on the database. Thus Dep# is restorable.

Nonessentiality Test

```
check_nonessent([],Rel,Attr) :- !.
check_nonessent([Key|KeysWith],Rel,Attr) :- rel_attrs(Rel,RelAttrs),closure_fd_ind(Rel,Key,Clos),
  ( sublist_check(RelAttrs,Clos),key(Rel,Key1),Key1\=Key -> true
  ; intersection(RelAttrs,Clos,Inter),remove(Attr,Inter,Left),reinsert_dep(Rel),
    ( closure_fd_ind(Rel,Left,Clos1),sublist_check(RelAttrs,Clos1) ->
      tnfdecomp(OrigRel,Rel),tnfdecomp(OrigRel,Rel1),Rel1\=Rel,
      key(Rel1,Key1),subset(Key1,Left),assert(key(Rel,Key1)),retract_dep(Rel)
    ; retract_dep(Rel),fail
    )
  ),!,check_nonessent(KeysWith,Rel,Attr).
```

As shown in the first clause of check_nonessent, if the attribute is not prime, then the attribute is nonessential. Otherwise, for each key Key containing Attr the closure Clos is computed. If all attributes RelAttrs of the relation are contained in the closure, then Attr is nonessential. Otherwise, the dependencies temporarily removed are reintroduced with reinsert_dep, Left is given by (Clos $\cap$ RelAttrs) $-$ {Attr} and the closure Clos1 of Left is computed. If all attributes RelAttrs are contained in Clos1, the attribute is nonessential and a key contained in Left must be added to the relation. Otherwise, the attribute is essential.

Remove the attribute

```
rem_attr(Rel,Attr) :- retract(remember_dep(Dep)),fail.
rem_attr(Rel,Attr) :- ind(SubRel,X,Rel,Y1),ind(Rel,Y2,SuperRel,Z),member(Attr,Y1),member(Attr,Y2),
  intersection(Y1,Y2,Inter),Inter\=[],corresp_attr(Y1,X,Inter,X1),corresp_attr(Y2,Z,Inter,Z1),
  not(ind(SubRel,X1,SuperRel,Z1)),assert(ind(SubRel,X1,SuperRel,Z1)),fail.
rem_attr(Rel,Attr) :- ind(Rel,X,SupRel,Y),remove(Attr,X,X1),retract(ind(Rel,X,SupRel,Y)),
  ( X1=[] -> true ; corresp_attr(X,Y,X1,Y1),assert(ind(Rel,X1,SupRel,Y1)) ),fail.
rem_attr(Rel,Attr) :- ind(SubRel,X,Rel,Y),remove(Attr,Y,Y1),retract(ind(SubRel,X,Rel,Y)),
  ( Y1=[] -> true ; corresp_attr(Y,X,Y1,X1),assert(ind(SubRel,X1,Rel,Y1)) ),fail.
rem_attr(Rel,Attr) :- key(Rel,Key),member(Attr,Key),retract(key(Rel,Key)),fail.
rem_attr(Rel,Attr) :- retract(superf(Rel,Attr)),retract(rel_attrs(Rel,RelAttrs)),
  remove(Attr,RelAttrs,RelAttrs1),assert(rel_attrs(Rel,RelAttrs1)).
```

Predicate rem_attr remove a superfluous attribute Attr of relation Rel from the database schema. It definitively removes the dependencies, adds the INDs that can be generated by transitivity involving attribute Attr of relation Rel, projects the INDs in which Attr appears, removes Attr from the relation schema, and removes all keys containing the attribute.

8 A Prolog-based Case Shell

This work has been carried out in a larger project in which a prototype CASE shell for Object-Oriented Information Systems Development is being constructed [29, 30]. The system, called *Yeroos Case*, is developed in LPA Prolog, and generates C++ code and relational database schemes in SQL. It integrates concepts and models from different object-oriented methods. In this paper we only described the ER+ and the Relational modules.

Figure 7 shows a printout of our system. ER+ conceptual schemas can be drawn directly on the screen using a Graphical User Interface. 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 introduced in the base of facts. These predicates form the input to the logical schema generator which automatically generates a relational schema normalized in IN-NF.

The use of Prolog for completely building our CASE environment is essential to our approach.

First, the declarative nature of Prolog offers several well-known advantages over conventional programming languages such as clarity, modularity, conciseness, and legibility. As a result, programs are much closer to the developer's perception of the domain and, for this reason, much easier to maintain. This is especially important in the context of large-scale programming.

Our system has a highly modular architecture to be able to manipulate different design methods other than the one explained here. The fact that Prolog is a dynamic language with untyped variables allowed us to implement different abstraction levels, realized by the separation of abstract and concrete primitives. *Abstract primitives* are low-level rules and primitives common to any module implementing a specific design method, while *concrete primitives* are rules specific to a particular module. Prolog allows us to define an integrated multi-layer architecture in which the abstract primitives are grouped in a kind of Operating System while the concrete primitives, grouped in concrete design method modules, are "plugged in" over this kind of OS.

Prolog offers a powerful and expressive development environment allowing to separate concepts from algorithmic control. To achieve representational independence, the fact base of our system consists of two repositories: one stores concepts and the other stores their graphical representation. These repositories were exploited by the Prolog inference engine, using the mechanisms of pattern-matching and backtracking.

Second, and more important, we use Prolog as the only language for constructing both the Graphical User Interface (GUI) and the core of the system. Our experience shows that, contrary to what is frequently believed, developing the entire application in Prolog is better and more consistent than the alternative approach of implementing the GUI in an imperative graphical development language such as Visual-Basic or Visual C++ and implementing the inference engine of the system in Prolog. This alternative approach is more complex, not uniform and sometimes not elegant: for instance, in such a dual approach in Windows, the Prolog inference engine must be called through Dynamic Link Libraries (DLLs) in order to be able to use it with the running GUI.

Also, the automatic memory management features of Prolog offer facilities for graphics and GUI development. In constrast with other imperative languages, Prolog does memory management automatically, freeing the programmer from this error-prone task. Thus, the developer always works with expressive names instead of low-levels pointers. This was a major benefit

when working with the fairly complex dynamic structures of graphical applications and sophisticated user interfaces.

Third, Prolog is better suited to CASE prototype design than conventional languages: its declarative nature encourages *incremental prototyping* where code is gradually fleshed out in a top-down manner over a period of time. Rather than throwing away the prototype and re-writing the final program from scratch, incremental prototyping saves development time and costs, and allows continuous user feedback. Imperative languages are not really suited for incremental prototyping. In this context, Prolog allows us to easily implement new models and methods but also to improve and update models and methods already implemented in the tool.

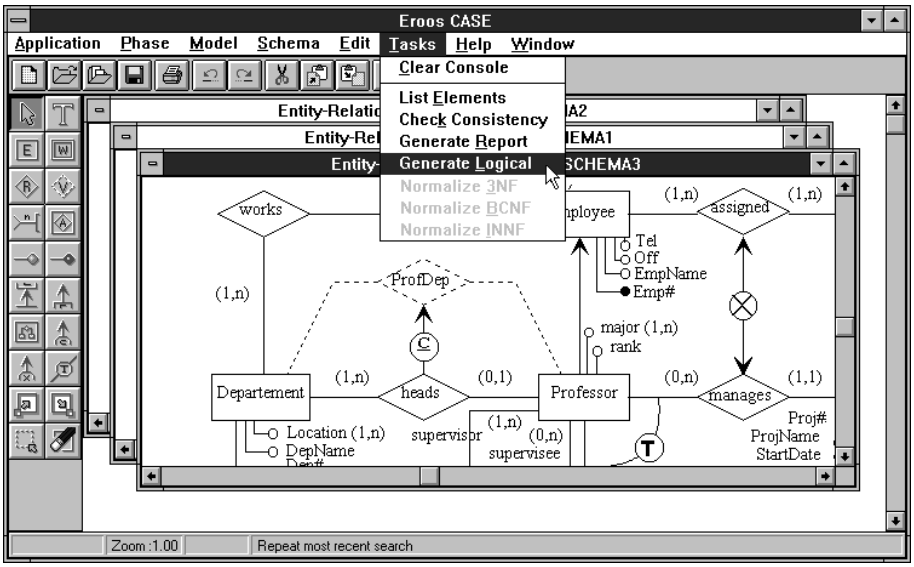

Figure 7: Layout of the Yeroos CASE.

9 Conclusion and Further Work

The main goal of our work has been to develop and test a method for relational database design based on [14], especially with respect to its viability in the context of computer-aided software engineering (CASE) shell development.

We demonstrated the usefulness of the inclusion normal 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.

We then presented our method for relational database design. It improves the algorithm of [14] in the following points:

- take into account multivalued attributes, weak entities and recursive relationships;
- distinguish generalization and subset constraints, in particular to be able to represent parallel generalizations;
- take into account several identifiers for entities, in particular due to the inheritance of identifers for (multiple) generalization;
- generate implicit FDs during the ER+ to relational mapping;
- use minimal covers instead of original sets of FDs;
- project the INDs for 3NF decompositions; and

- generate INDs that can be deduced by transitivity, before removing superfluous attributes, and specialize INDs after removing each superfluous attribute.

Several databases have been tested and have provided convincing results. These tests 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 detect errors early in the development life cycle, which constitutes a necessity in software engineering. Our system validites 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 to automatically generate the corresponding relational database schema, normalized in IN-NF.

Several issues need to be further investigated. Concerning the ER+ formalism, other abstractions like materialization [22] could also be implemented.

Our mapping rules produce relations keeping track of the distinction between entities and relationships. Optimized rules can generate fewer relations but lose this semantic classification. Our system could be used to compare pros and cons of each method. Optimization of our mapping rules might be realized by implementing a relation merging algorithm [20].

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 [27]. Every relation of a database that is in IN-NF is only guaranteed to be in 3NF. However, as it 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. In *Proc. of the IFIP Congress*, pages 580–583, Geneva, Switzerland, 1974.

[2] P. Atzeni and E. Chan. Independent database schemes under functional and inclusion dependencies. In P. Stocker, W. Kent, and P. Hammersley, editors, *Proc. of the 13th Int. Conf. on Very Large Data Bases, VLDB'87*, pages 159–166, Brighton, England, 1987. Morgan Kaufmann.

[3] C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: An Entity-Relationship Approach.* Benjamin/Cummings, 1992.

[4] C. Beeri, P. Bernstein, and N. Goodman. A sophisticate's introduction to database normalisation theory. In S. Yao, editor, *Proc. of the 4th Int. Conf. on Very Large Data Bases, VLDB'78*, pages 113–124, West Berlin, Germany, 1978. Morgan Kaufmann.

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

[6] M. Bouzeghoub, G. Gardarin, and E. Metais. Database design tools : An expert system approach. In A. Pirotte and Y. Vassiliou, editors, *Proc. of the 11th Int. Conf. on Very Large Data Bases, VLDB'85*, pages 82–95, Stockholm, Sweden, 1985. Morgan Kaufmann.

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

[8] M. Casanova, L. Tucherman, A. Furtado, and A. Braga. Optimization of relational schemas containing inclusion dependencies. In P. Apers and G. Wiederhold, editors, *Proc. of the 15th Int. Conf.*

on Very Large Data Bases, VLDB'89, pages 317–325, Amsterdam, The Netherlands, 1989. Morgan Kaufmann.

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

[10] E. Codd. Further normalization of the database relational model. *Data Base Systems*, 6:33–64, 1972.

[11] A. Doğaç, B. Yuruten, and S. Spaccapietra. A generalized expert system for database design. *IEEE Trans. on Software Engineering*, 15(4):479–491, Apr. 1989.

[12] R. Elmasri and S. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, 2 edition, 1994.

[13] R. Fagin and M. Vardi. The theory of data dependencies. In M. Anshel and G. Gerwitz, editors, *Mathematics of Information Processing*, pages 19–72. 1986.

[14] C. Goh. Towards a viable methodology for logical relational database design. Master's thesis, National University of Singapour, 1992.

[15] F. Golshani, editor. *Proc. of the 8th Int. Conf. on Data Engineering, ICDE'92*, Tempe, Arizona, 1992. IEEE Computer Society.

[16] M. Kolp and E. Zimányi. Relational database design using an ER approach and Prolog. In S. Bhalla, editor, *Proc. of the 6th Int. Conf. on Information Systems and Management of Data, CISMOD'95*, LNCS 1006, pages 214–231, Bombay, India, Nov. 1995. Springer-Verlag.

[17] T. Ling and C. Goh. Logical database design with inclusion dependencies. In Golshani [15], pages 642–649.

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

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

[20] M. Markowitz. Merging relations in relational databases. In Golshani [15], pages 428–437.

[21] J. Mitchell. Inferences rules for functional and inclusion dependencies. In *Proc. 2nd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, PODS'83*, pages 58–69, 1983.

[22] A. Pirotte, E. Zimányi, D. Massart, and T. Yakusheva. Materialization: a powerful and ubiquitous abstraction pattern. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. of the 20th Int. Conf. on Very Large Data Bases, VLDB'94*, pages 630–641, Santiago, Chile, 1994. Morgan Kaufmann.

[23] W. Potter. A multi-model schema design tool in Prolog. In *Proc. of the 1st Int. Conf. on Expert Database Systems*, pages 747–759, 1984.

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

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

[26] T. Teorey. *Database Modeling and Design: The Entity-Relationship Approach.* Morgan Kaufmann, 1990.

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

[28] J. Ullman. *Principles of Database Systems.* Computer Science Press, 1982.

[29] E. Zimányi and M. Kolp. Using Prolog to implement a CASE shell for object-oriented development. In *Proc. of the 8th Symposium and Exhibition on Industrial Application of Prolog, INAP'95*, pages 41–48, Tokyo, Japan, Oct. 1995.

[30] E. Zimányi and M. Kolp. A Prolog-based architecture for an object-oriented CASE shell. In *Proc. of the 4th Int. Conf. on Practical Application of Prolog, PAP'96*, pages 497–520, London, UK, Apr. 1996.