A Methodology and Tool for Top-down Relational Database Design

Naphtali Rishe

School of Computer Science
Florida International University
University Park, Miami, FL 33199

This paper presents a methodology for logical design of relational schemas and integrity constraints using semantic binary schemas. This is a top-down methodology. In this methodology, a conceptual description of an enterprise is designed using a semantic binary model. Then, this description is converted into the relational database design.

The paper also describes a tool which automates all the busy work of the methodology and provides graphic output. With respect to the intelligent design decisions, the tool accepts instructions from its user, who is a database designer, or, when the user defaults, makes decisions itself based on "rule-of-thumb" principles.

Keywords: database design, relational databases, semantic databases, CASE tools, methodology, database schemas, integrity constraints

---

1. INTRODUCTION

In the methodology presented in this paper, semantic binary schemas are converted into relational schemas and integrity constraints. The semantic database models offer a simple, natural, implementation-independent, flexible, and non-redundant specification of information and its semantic aspects. Since the original idea of [Abrial-74], many semantic data models have been studied in the Computer Science literature. Many semantic models have been surveyed in [Hull&King-87] and [Peckham&Maryanski-88]. Although somewhat differing in their terminology and their selection of tools used to describe the semantics of the real world, the various semantic models are roughly equivalent. This paper's methodology uses the Semantic Binary Model (SBM) ([Rishe-92-DDS], [Rishe-87-RM], [Rishe-88-DDF], [Rishe-89-SD], [Rishe-90-SB]), a descendant of the model of [Abrial-74]. Models similar to SBM have been studied in: [Bracchi&*al.*-76], [Nijssen-77], [Nijssen-81], [Breutman&*al.*-79], [Senko-78], [Mark-83], [Mark-87], [Mark-89], [Meersman&Assche-83], [Vermeir-83], and others. SBM does not have as rich an arsenal of tools for semantic description as can be found in some other semantic models, e.g. the IFO model [Abiteboul&Hull-84], SDM [Hammer&McLeod-81], the Functional Model ([Kerschberg&Pacheco-76], [Shipman-81], [Chan&*al.*-82]), SEMBASE [King-84], NIAM ([Nijssen-81], [Verheijen&VanBekkum-82], [Leung&Nijssen-87]), Taxis [Nixon&*al.*-87], SIM [Jagannathan&*al.*-88], SAM [Su&Lo-80], OSAM* [Su-88-OS], GEM [Tsur&Zaniolo-84], GENESIS [Batory&*al.*-88], ER [Chen-76], Extended ER [Batini&*al.*-92]. Nevertheless, the SBM has a small set of sufficient simple tools by which all of the semantic descriptors of the other models can be constructed.

Non-procedural languages for semantic databases have been studied in [Meersman-81], [Shipman-81], [Senko-78], [Rishe-86-PS], [Rishe-91-PC], and others. Extended Pascal data manipulation language for semantic databases is defined in [Rishe-88-TM] and [Rishe-88-DDF]. Data definition languages integrated with data manipulation languages for semantic databases have been studied in [Roussopoulos&Mark-85].

The use of semantic models for the design of relational schemas has been studied in [Brodie&*al.*-84-CM], [Chen-76], [King&McLeod-85], [Teorey&*al.*-86], [Leung&Nijssen-87], [Shoval-85], [Shoval/Even-Chaime-87], [Verheijen&VanBekkum-82], [DeTroyer&Meersman-86], [Rosenthal&Reiner-90], and other works. A graphical interactive system for the design of semantic databases is discussed in [Shoval&*al.*-88].

The methodology proposed herein differs in the satisfaction of a broad range of schema-quality criteria (to be introduced); in comparative analysis of different design choices in various steps of design; and in systematic generation of integrity constraints. Among the novel points in the methodology are procedures for the generation of keys of categories and for partitioning of non-disjoint categories into disjoint ones. The treatment of sub-super categories and non-disjoint categories is important for the proper reflection of the original semantics in the resultant relational schema and for avoidance of logical redundancy of information in the database. Other methodologies (e.g. [Shoval/Even-Chaime-87], [Verheijen&VanBekkum-82], [Rosenthal&Reiner-90]) do not address this issue explicitly. On the other hand, they do address other issues that the methodology of the present paper does not consider directly. In the input semantic model of this methodology, SBM, the semantic issues that are rather simple to the user are graphically explicit. Other semantic nuances are relegated to integrity constraints, and they are propagated into the relational schema's external integrity constraints. For example, the input model of [Shoval/Even-

Chaime-87] allows nested relationships.

This paper begins with a brief description of the semantic binary model. Objectives of schema design follow. Then the methodology is presented. Then an automatic tool is described. The last section compares this methodology to the normalization methodology.

2.   THE SEMANTIC BINARY MODEL

This section describes the Semantic Binary Model.

The semantic binary database model represents information of an application's world as a collection of elementary facts of two types: unary facts categorizing objects of the real world and binary facts establishing relationships of various kinds between pairs of objects.

One of the major advantages of the relational database model, as compared to the network and hierarchical models, was the independence of the logical *data* from the physical aspects of data storage. The semantic models went one step forward towards the independence of the actual *information* from its logical data representation. Among the semantic advantages of the semantic binary model relative to the relational model are the following:

- All the information is composed of the elementary facts describing the real world, so no normalization of a semantic binary schema is needed;

- No category (type) of objects needs to have a key. A key is collection of attributes which are never *null* and which universally identify the objects of the category. (Instead of having a fixed inflexible key, in the semantic model different objects of the category may be identifiable by different attributes or by different relationships with objects of perhaps other categories. In the real world, keys exist very rarely. Another practical requirement of the keys in most systems is that their values do not change in time. (Even if the system does not guard keys against change, arbitrary changes may cause inconsistency of the database.) On the other hand, all natural attributes should be allowed change in time: if not because of the changes in the real world then because of correction of erroneously recorded data.)

- Objects are not logically replaced by their keys, when these exist. So a value of a key is changeable with no influence on the other information about this object in the database.

- An object may belong to several categories simultaneously.

- Properties which are common to several categories, can be specified just once.

- It is conceptually simple and schemas can be easily explained to owners of the information to be stored in the database, who may have no computer knowledge but must approve the conceptual schema.

A definition of the model's concepts follows.

2.1. Categories

Object — any item in the real world. It can be either a concrete object or an abstract object as follows.

Value, or **Concrete Object** — a printable object, such as a number, a character string, or a date. A value can be roughly considered as representing itself in the computer, or in any formal system.

Abstract Object — a non-value object in the real world. An abstract object can be, for example, a tangible item (such as a person, a table, a country), or an event (such as an offering of a course by an instructor), or an idea (such as a course). Abstract objects cannot be represented directly in the computer.

This term is also used for a user-transparent representation of such an object in the Semantic Binary Model.

Category (also called *Entity Type* or *Entity Set* in some semantic models) — any concept of the application's real world which is used for classification of objects. At every moment in time such a concept is descriptive of a set of objects.

Unlike the mathematical notion of a set, the category itself does not depend on its objects: the objects are added and removed while the meaning of the category is preserved in time. Conversely, a set *does* depend on its members: the meaning of a set changes with the ebb and flow of its members.

Categories are usually named by *singular* nouns. An object may belong to several categories at the same time.

Disjoint categories — Two categories are *disjoint* if no object may simultaneously be a member of both categories. This means that at every point in time the sets of objects corresponding to two disjoint categories have empty intersection.

Subcategory — A category is a *subcategory* of another category if at every point in time every object of the former category should also belong to the latter. This means that at every point in time the set of objects corresponding to a category contains the set of objects corresponding to any subcategory of the category.

Abstract category — a category whose objects are always abstract.

Concrete category, **category of values** — a category whose objects are always concrete.
Many concrete categories, such as *NUMBER, STRING,* and *BOOLEAN,* have constant-in-time sets of objects. Thus, those concrete categories are actually indistinguishable from the corresponding sets of all numbers, all strings, and the Boolean values ({TRUE, FALSE}).

2.2. Binary Relationships

Binary Relationship — any concept of the application's real world which is a binary property of objects, that is, the meaning of a relationship or connection between two objects. At every moment in time, the relationship is descriptive of a set of pairs of objects which are related at that time. The meaning of the relationship remains unaltered in time, while the sets of pairs of objects corresponding to the relationship may differ from time to time, when some pairs of objects cease or begin to be connected by the relationship.

Notation: "$x R y$" means that object x is related by the relationship R to object y.

Types of binary relationships:

- A binary relationship R is **many-to-one** (**m:1**, **functional**) if at no point in time xRy and xRz where $y \neq z$.

- A binary relationship R is **one-to-many** (**1:m**) if at no point in time xRy and zRy where $x \neq z$.

- Relationships which are of neither of the above types are called **proper many-to-many** (**m:m**).

- A binary relationship which is both *m:1* and *1:m* (always) is called **one-to-one** (**1:1**).

- A binary relationship is **proper m:1** if it is m:1 and not 1:1.

- A binary relationship is **proper 1:m** if it is 1:m and not 1:1.

Domain and **range** of a binary relationship:

A category C is the domain of R if it satisfies the following two conditions:

a. whenever xRy then x belongs to C (at every point in time for every pair of objects); and

b. no proper subcategory of C satisfies (a).

A category C is the range of R if:

a. whenever xRy then y belongs to C (at every point in time for every pair of objects); and

b. no proper subcategory of C satisfies (a).

Total binary relationship — A relationship R whose domain is C is *total* if *at all times* for every object x in C there exists an object y such that xRy. (At different times different objects y may be related to a given object x.)

Note: No relationship needs to be *total* on its domain.

2.3. Non-binary Relationships

Non-binary relationships — real-world relationships that bind more than two objects in different roles. Such complex relationships are regarded in the Binary Model as groups of several simple relationships.

In general, the Binary Model represents any non-binary relationship as:

a. An abstract category of events. Each event symbolizes the existence of a relationship between a group of objects.

b. Functional binary relationships, whose domain is the category (a). Each of those functional binary relationships corresponds to a role played by some objects in the non-binary relationship.

Thus, the fact that objects $x_1, \ldots, x_n$ participate in an n-ary relationship R in roles $R_1, \ldots, R_n$, is represented by:

a. an object e in the category R', and

b. binary relationships $eR_1 x_1, \ldots, eR_n x_n$.

2.4. Graphic Representation of Schemas

1. In a schema, abstract categories are shown by rectangles.

2. Relationships from abstract categories to concrete categories are shown inside the boxes of the domain-categories as follows:

<div align="center">

relationship **:** *range type*

</div>

The *range* is specified as a programming language data-type. (We will use the style of *Pascal* here.)

Usually, relationships between abstract and concrete categories are *m:1*. This is the *default type of relationships whose ranges are concrete categories*, and it need not be explicitly specified in the schema for such relationships.

3. Relationships between abstract categories are shown by arrows between the categories' rectangles. (The direction of the arrow is from the domain to the range.) The name and type of the relationship are indicated on the arrow. The *default for the type of relationships between abstract categories* is *m:m*.

4. Subcategories are connected to their supercategories by arrows with dashes.

5. The disjointness of categories is indicated implicitly:

a. Two categories which have a subcategory in common are *not* disjoint. (The common subcategory does not have to be their *immediate* subcategory, that is, it may be a subcategory of a subcategory, and so on.)

b. Two categories which are subcategories of one category (not necessarily immediate subcategories) are considered *not* disjoint, unless otherwise declared in an appendix to the schema.

c. The other categories are disjoint from each other, unless otherwise declared in an appendix to the schema.

Figure 2-1 is an example of a semantic binary schema.
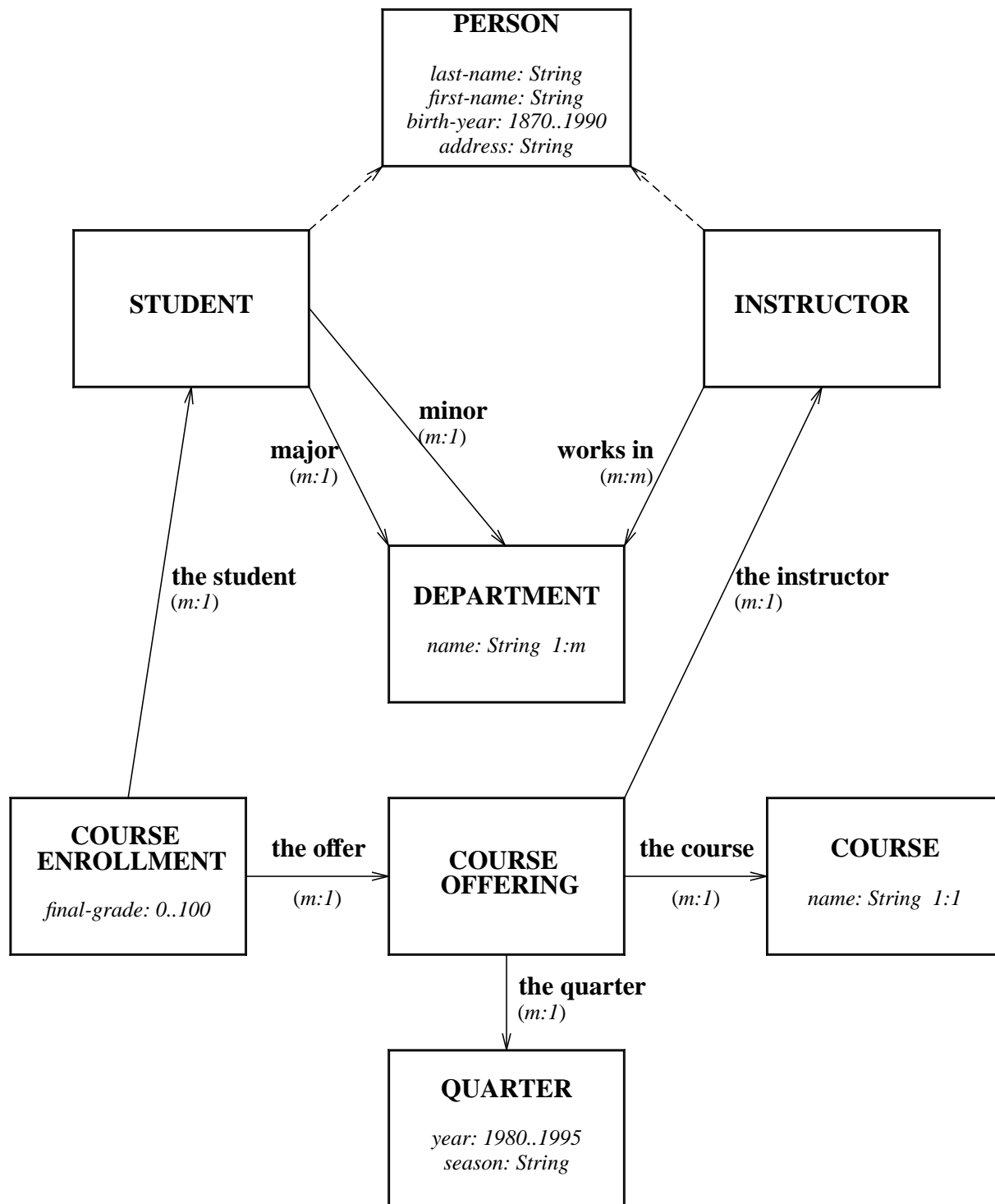
Figure 2-1. A semantic schema for a university application. Boxes are abstract categories with their attributes (concrete relations). *STUDENT* and

INSTRUCTOR are nondisjoint subcategories of *PERSON*. The relation *works-in* is many-to-many.

3.   THE RELATIONAL MODEL

For convenience of the database design and use of languages, this section defines the Relational Model as a subset of the Semantic Binary Model.

Attribute — A functional relationship whose range is a concrete category.

> The phrase '*a* is an **attribute of** *C* ' means:
> *a* is an attribute, and its domain is the category *C*.

Time-invariant attribute — An attribute *A* is *time-invariant* if once an object *x* becomes related by *A* to a value *y*, the object *x* will forever be related by *A* to *y*, as long as *x* exists.

> There are no time-invariant attributes in the natural user world. Even if the laws of physics or society do not allow for an attribute to change in time, the attribute may change in the perceived real world due to discoveries of errors in earlier perception. For example, a social security number could be wrongly reported and then corrected. Thus, *time-invariance* is defined only in implementational restrictions. Such restrictions are unavoidable in the relational database design. The methodology of relational schema design that is presented below has among its goals the minimization of the negative effect of such implementational restrictions.

Keys

1. **Single-attribute key**

   A time invariant attribute of a category is called its *key* if it is *1:1* and *total*. That means that the values of the attribute can be used to identify the objects of the category.

   Due to the *time-invariance* requirement, no attribute is really a key in the natural user's world. Thus, the property of a *key* is defined only in implementational restrictions, which are unavoidable in the relational database design. Also, the requirement of *totality* is very rarely an integrity constraint imposed by the logic of the user world, but rather is an implementational restriction.

Convention: In this paper, we shall name the attributes constrained to be keys with the suffix *-key*.

2. **Multi-attribute key**

   The following definition extends the concept *key* to a collection of attributes.

Key of a category — a *collection* of total time-invariant attributes $f_1, f_2, \ldots, f_n$ whose domain is that category s.t.

(i)   For any collection of values, $x_1, \ldots, x_n$ there is no more than one object *y* of the category s.t.

$$x_1 = y.f_1 \text{ and } x_2 = y.f_2 \text{ and } \cdots \text{ and } x_n = y.f_n$$

(ii)   No proper subcollection of these attributes *always* satisfies (i).

Practically, requirement (i) means that the collection of attributes is sufficient to identify every object of the category. Requirement (ii) means that the collection is minimal: if one of the attributes is not known then the remaining attributes might not provide sufficient information to identify every object of the category.

Convention: in this paper, when a category is constrained to have exactly one key, and the key is composed of several attributes, we shall name these attributes with the suffix *-in-key*.
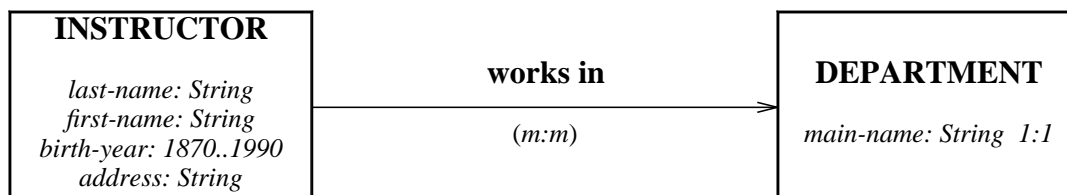
Note :

(i)   In the real world a category usually has *no* key. Thus, the existence of a key is usually not an integrity constraint but rather an implementational restriction. This restriction will be imposed when unavoidable due to limitations of a DBMS or a database model, especially the relational model.

(ii)   Existence of a key makes every object of the category identifiable with the values of the key and eliminates the necessity to refer to abstract objects.

(iii) A category which has no key may still have all its objects completely identifiable (using different relationships and their combinations for different objects), but the identification would not be uniform.

(iv)   When a key is composed of several attributes it is still *one* key.

(v)   A category may theoretically have several keys. However, since categories in the real world rarely have even one key, the existence of more than one key would be an unnecessarily strong implementational restriction (particularly because the time-invariance requirement), which is not required by database management systems. The possibility of more than one key (called *candidate keys* in some literature) is irrelevant to the purposes of this paper. In the design algorithm we start with no keys (which is a natural state) and create one key per category (which suffices for the relational schema).

A binary schema is called **table-oriented**, or a **relational schema** if

(i)   all the abstract categories of the schema have keys

(ii)   all the abstract categories are pairwise disjoint
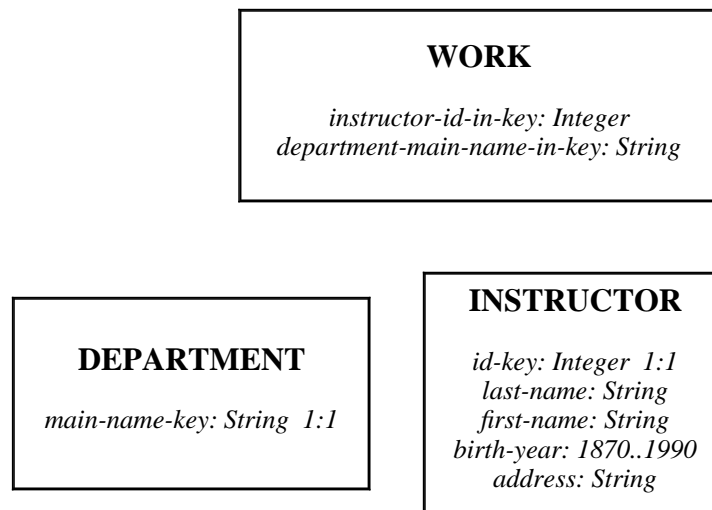
(iii) the only relationships are attributes.

Thus, all the information in a relational schema is represented by attributes of categories.

Figure 3-1. A semantic schema.

WORK

instructor-id-in-key: Integer
department-main-name-in-key: String

DEPARTMENT

main-name-key: String 1:1

INSTRUCTOR

id-key: Integer 1:1
last-name: String
first-name: String
birth-year: 1870..1990
address: String

Figure 3-2. A relational schema.

4. DESIGN GOALS

4.1. Quality of Schemas

A schema is said to be of *high quality* if it satisfies the following criteria (*cf.* Chapter 2 of [Batini&*al.*-92].)

1. The schema describes the concepts of its application's world *naturally*:

    • The schema describes the objects, categories and relationships as they are in the real world.

    • The users can translate ideas easily in both directions between the concepts of the schema and the natural concepts of the application world.

2. The schema contains very little or *no* **redundancy**.

    The redundancy should be avoided *not* in order to improve the storage efficiency — the storage is not that expensive nowadays.

    The redundancy should be avoided primarily in order to prevent inconsistency of the database and its update anomalies.

    When the redundancy is needed for the convenience of the users, it should be introduced into the *user-views* but not into the schema.

In some database models we cannot eliminate the redundancy completely. When we have to have some redundancy, we should at least bind it by integrity constraints. When such constraints are implemented, the user is forced to update all the related facts simultaneously.

3. The schema does *not impose implementational restrictions*, that is, every situation probable in the real world of the application is fully representable under the schema.

4. The schema covers by itself as many **integrity constraints** as possible, that is, the class of instantaneous databases formally possible according to the schema is not much larger than the class of all possible situations of the real world.

   Constraints that are *not* expressed in the schema cause these problems:

   • They are hard to formulate and to specify.

   • They are seldom enforced by the DBMS. Thus, they require a substantial application programming effort for their enforcement, are often implemented incorrectly, and usually prevent direct interaction between the user updating the database and the DBMS (the user may not use the standard language for simple updates, which is supplied by most DBMS).

   • The users and application programmers often forget or misunderstand such constraints.

5. The schema is **flexible**: if probable changes in the *concepts* of the application world occur, the schema would not have to undergo drastic changes.

6. The schema is **conceptually-minimal**: it does not involve concepts which are irrelevant in the application's real world, and limits the accumulation of information which is irrelevant in that world.

The most important issue of the database design is the design of a high-quality schema within the restrictions of the available DBMS and database model. A low-quality schema increases the chances of corruption of the data, makes it very hard to use and maintain the database, and makes it very hard, if not impossible, to adjust the database to the changing concepts of the application's real world.

It is easy to design a high quality schema in semantic models, particularly the Semantic Binary Model. The task is much harder in the Relational Model. Moreover, it is usually impossible to describe an application world by a schema in the Relational Model with the same high quality as with which that application can be described in the Semantic Binary Model.

The objective of this paper is to make the task of design of a relational schema easy once a semantic binary schema has been designed.

4.2. Objectives of Schema Conversion

A **Schema-conversion** is a replacement of a schema by another schema having the same information content. This means that each of the two schemas can be regarded as a user-view of the other.

Schema-conversion is a means of database design: a schema is first designed in a higher-level database model and then translated into a lower-level model which is supported by the available DBMS (when a DBMS for a higher-level model is unavailable or inadequate). This paper presents a conversion algorithm of an adequate binary schema into a relational schema whose quality is among the highest possible for the Relational Model.

The quality of the resulting schema is assessed with respect to the above criteria.

This algorithm can be performed manually by the database designer. Alternatively, an automatic tool can be used to perform all the busy work, while prompting the database designer for intelligent decisions (and using defaults when the designer fails to provide such a guidance). We have developed such a tool.

5.   THE CONVERSION ALGORITHM: SBM TO RELATIONAL

In this paper, the constraints are specified in a form of first-order predicate calculus adapted to databases. A full description of this language is given in [Rishe-88-DDF].

5.1.   Composition and Split of Relationships

Two auxiliary definitions of terminology that will be used in the conversion algorithm follow.

Composition of relationships

The definition is preceded by an example.

> *Example 5-1.*
>
> Consider two relationships:
>
> ☐ *the-course* — relationship from *COURSE-OFFERING* to *COURSE* (*m:1*)
>
> ☐ *name* — relationship from *COURSE* to *String*   (*1:1*)
>
> The composition of *THE-COURSE* on *NAME* is:
>
> ☐ *the-name-of-the-course* — relationship from *COURSE-OFFERING* to *String*   (*m:1*)

Let the range of Relationship R_1 be the domain of Relationship R_2 (Figure 5-1). (This domain/range is denoted C in the figure; in the above example it is the category *COURSE*.)

Relationship R (*the-name-of-the-course* in the above example) is the composition of R_1 (*the-course* in the example) and R_2 (*name* in the example) if:

for every x, y, xRy iff there exists z such that xR_1z and zR_2y.
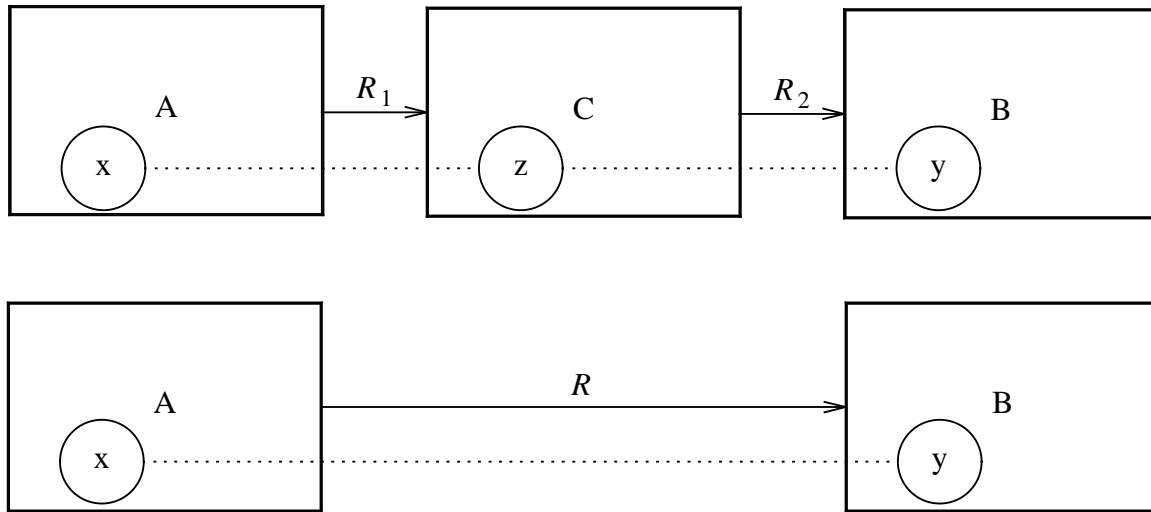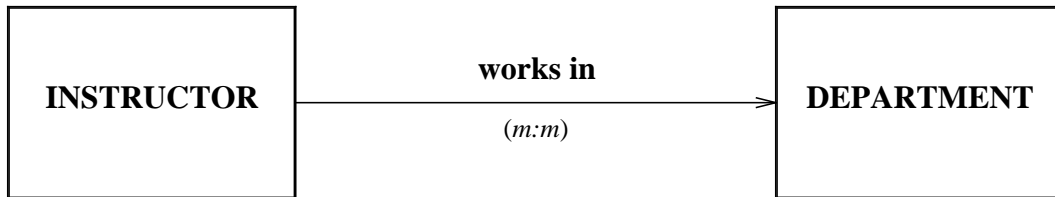

Figure 5-1. Relationship R is the composition of two relationships, R_1 and R_2: xRy is whenever xR_1z and zR_2y.

The following example precedes the definition of another operation, *relation-split*.

Figure 5-2. A m:m relationship.

Example 5-2.

If due to an implementational restriction we may not have a *m:m* relationship of Figure 5-2 then we can split it into a new category with two relations as in Figure 5-3.

This split necessitates additional integrity constraints:

(i) Both new relationships are total.

(ii) For any combination of an instructor and a department there is at most one object in work.

The latter constraint is more rigorously formulated in calculus, as follows.

for every w **in** *WORK*:

    for every v **in** *WORK*:

        if w.*the-instructor* = v.*the-instructor* **and**
            w.*the-department* = v.*the-department*
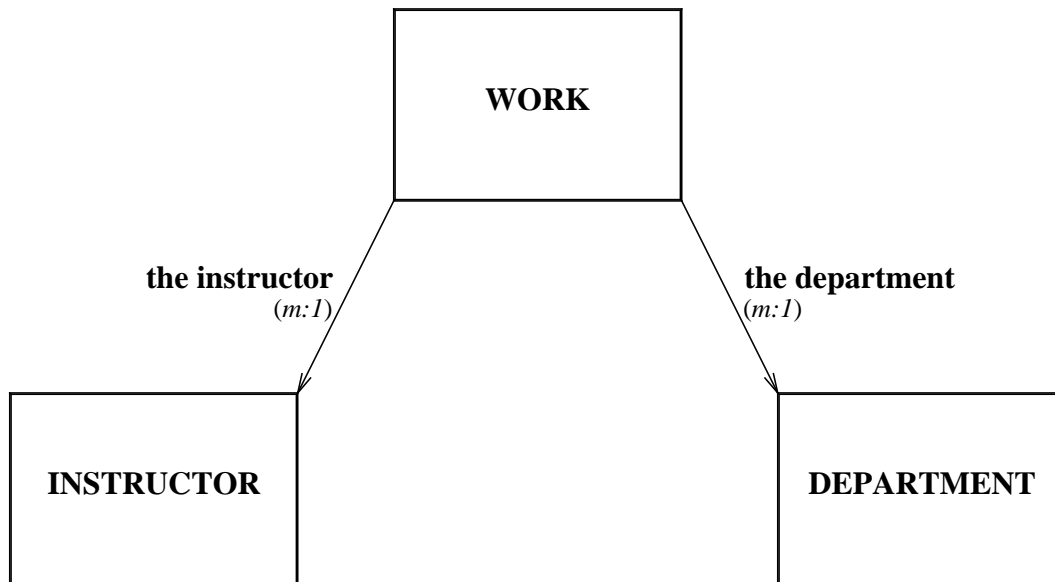
Figure 5-3. Split of a m:m relation.

Relationship-split — conversion of a schema having a relationship R into another schema having, instead of R, a new abstract category C and two total functional relationships R_1, R_2, whose domain is C, s.t. xRy *iff* there exists an object z in C for which zR_1x and zR_2y. (Figure 5-4.) (In the above example, R is the many-to-many relation *works-in* and C is the new category *WORK*.)
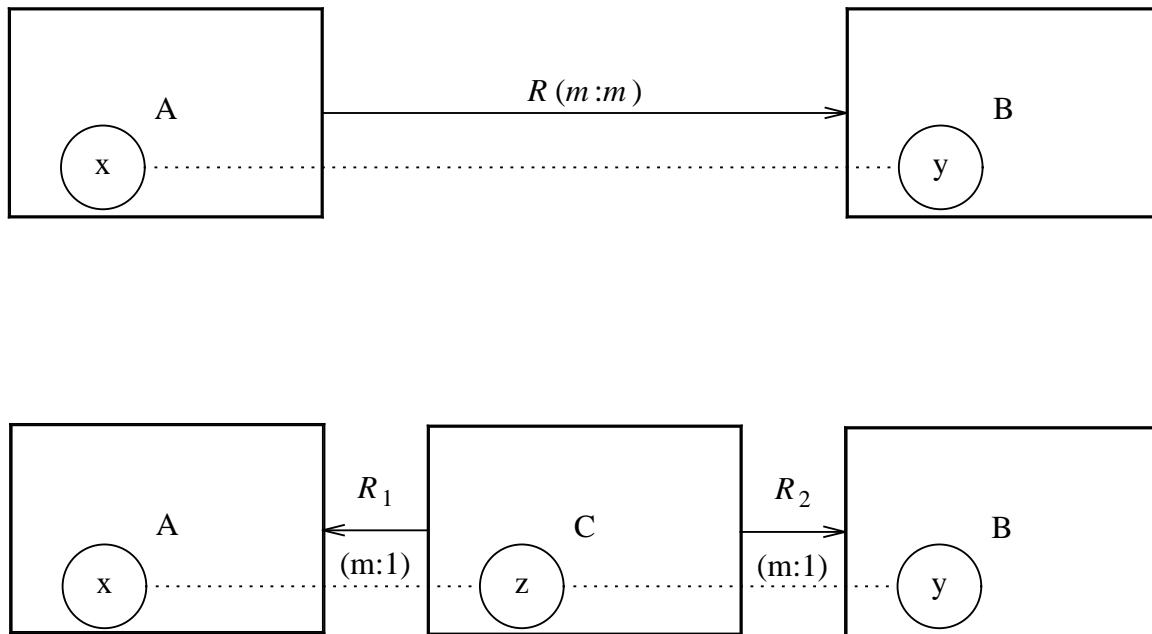
Figure 5-4. Relationship R is split into a category C and two relationships, R_1 and R_2. Every connection $x-y$ is broken into $x-z$ and $z-y$.

The following subsections present the conversion algorithm

5.2. Keys

(The concept of *key* was defined in Section 3.)

Step 1. Choose a key for every abstract category, excluding subcategories of other categories, as follows.

    a.   (**single-attribute key**)

       if the category has an attribute which is *1:1*, time-invariant, and total, *then* let that attribute be the key;

    b.   (''forced'' **single-attribute key**)

       else if the category has an attribute which can be implementationally restricted to be *1:1*, time-invariant, and total, without very harmful alteration of the real world, *then* make that attribute into a key (declare the implementational restriction);

Example 5-3.

<div align="center">

name-key of *COURSE*

</div>

(Please note that according to our convention specified *supra* , the names of key attributes have a suffix *-key*.)

It is not a very far reaching alteration of the real world to make this implementation restriction: "Every course has exactly one name, and this name may never be changed."

c. (**multi-attribute key**)

else if the category has a collection of attributes which are time-invariant and total, and jointly identify all the objects in the category, *then* let a minimal such collection be the key;

Example 5-4.

(*season-in-key, year-in-key*) of *QUARTER*

(Please note that according to our convention specified *supra*, the names of attributes that are parts of multi-attribute keys have a suffix *-in-key*.)

d. (''forced'' **multi-attribute key**)

else if the category has a collection of attributes which can be implementationally restricted to be time-invariant and total, and to jointly identify all the objects of the category, without very harmful alteration of the real world, *then* make a minimal such collection of attributes into a key;

e. (**inferred key**)

else if a collection of attributes can be inferred from the information existing in the schema and from keys of other categories, so that

- these attributes can be implementationally restricted, without very harmful alteration of the real world,
  (i) to be time-invariant and total, and
  (ii) to jointly identify all the objects of the category,

then

(i) choose a minimal such collection of inferrable attributes;

(ii) add to the schema those attributes from the collection which are not already in the schema;

(iii) make this collection of attributes into a key (declare the implementational restrictions);

(iv) convert the inference rule of these attributes into constraints. (Since these will now be new attributes, their values will be updated by the users with possible inconsistency relative to the information from which these attributes are inferrable.)

Example 5-5.

To obtain a key for *DEPARTMENT* we alter the real world slightly: we require

every department to have at least one name; we shall call the first name ever given to a department the ''*main-name*'', and we require that the *main-name* of a department may never be changed. We add the new attribute

□ *main-name-key* — attribute of *DEPARTMENT*, range: *String* (*1:1*)

and the constraint

for every d **in** *DEPARTMENT:*

d *NAME* d.*MAIN-NAME-key*

Note: In conjunction with the implicit constraint *-key*, the above constraint means that the main-name is the first name ever given to the department, and that it will remain the department's name forever.

Example 5-6.

More characteristic examples of inferred keys are for the categories *COURSE-OFFERING* and *COURSE-ENROLLMENT*. These will be given and generalized after we have a key for *PERSON*.

---

f. (**enumerator id key**)

else create a new external enumeration for the objects in the category (thus altering the real world) and add it as an attribute, which will be the chosen key.

Example 5-7.

The key of *PERSON* will be a new attribute *id-key*.

Pragmatically, a program should be written to generate new values of an *enumerator id key*. These numbers will be assigned by the user to the new objects of the category. The numbers may not be reused when an object is removed. The numbers themselves should bear no correlation to the other information in the database, since the other information may change in time, while the key is time-invariant.

It is also advisable that the numbers be not assigned sequentially, but rather in an arbitrary sequence. Otherwise, the irrelevant information on the ''seniority'' of objects will be hidden in the id. Any hidden information will be abused by the application programmers. Since it is not always possible to update such hidden information correctly, the programs will not produce the expected results in some special cases.

Note: The step of finding keys is performed simultaneously for all the categories, since we might need to know the key of one category in order to find a key of a related category.

Example 5-8.

An *inferred key* of *COURSE-OFFERING* can be obtained when keys for *QUARTER*, *COURSE*, and *PERSON* have been chosen. The inferred key of *COURSE-OFFERING* will be

> {the name of the course, the year of the quarter, the season of the quarter, id of the instructor}

Hence, we add 4 new attributes to *COURSE-OFFERING*. The *inferred* key of *COURSE-ENROLLMENT* will be 5 new attributes

> {id of the student, the key of the offering}.

(The "key of the offering" consists of four attributes. Thus, there is a total of five attributes in the key of *COURSE-ENROLLMENT*.)

Example 5-9.

The category *COURSE-OFFERING* is now as in Figure 5-5.

COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
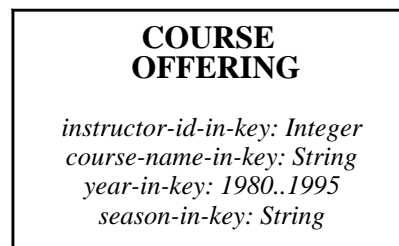
Figure 5-5. The revised category
COURSE-OFFERING

The above is an example of the prevalent case of an *inferred key*. The following is a generalization of this example.

Assume that a category C is the domain of total functional relationships $f_1, \ldots, f_n$ which jointly identify all the objects of the category.

Example 5-10.

Every course offering is uniquely identified by its instructor, course, and quarter. Thus, the total functional relationships

> *THE-INSTRUCTOR, THE-COURSE, THE-QUARTER*

jointly identify all the objects of their domain, the category *COURSE-OFFERING*.

The above assumption means that there is an integrity constraint

for every x **in** C:

    for every y **in** C:

        if x.f_1=y.f_1 **and** $\cdots$ **and** x.f_n=y.f_n

            then x=y

Example 5-11.

for every x **in** *COURSE-OFFERING*:

    for every y **in** *COURSE-OFFERING*:

        if

            x.*THE-INSTRUCTOR*=y.*THE-INSTRUCTOR* **and**

            x.*THE-COURSE*=y.*THE-COURSE* **and**

            x.*THE-QUARTER*=y.*THE-QUARTER*

        then x=y

In this case, once the keys of the ranges of the functional relationships $f_1, \ldots, f_n$ are known, a key of C can be inferred from them. Let the keys of the ranges be $k_1, \ldots, k_n$. Let k_i-of-f_i be the set of inferred attributes obtained by the composition of the attributes comprising the key k_i and the relationship f_i.

Example 5-12.

There are three such sets of inferred attributes for the category *COURSE-OFFERING*:

- id-of-the-instructor

- the-name-of-the-course

- the-year-of-the-quarter, the-season-of-the-quarter

The key of C is contained in the union of compositions of the relationships f_i onto the keys of their ranges, that is,

$$\{(k_1 \text{ of } f_1), \cdots, (k_n \text{ of } f_n)\}$$

Notice that the key of C is *contained* in the above union of compositions. Usually the key of C is equal to that union of compositions, but sometimes it is *properly* contained.

Example 5-13.

Let us change the meaning of *COURSE-OFFERING*. Now, it does not have to occur in one particular quarter, but can last several quarters, as long as the quarters are within one academic year. There are two relationships between

---

offerings and quarters:

- ☐ *beginning-quarter* — relationship from *COURSE-OFFERING* to *QUARTER* (*m:1,total*)

- ☐ *ending-quarter* — relationship from *COURSE-OFFERING* to *QUARTER* (*m:1,total*)

The key of *COURSE-OFFERING* is properly contained in

{the name of the course;
the year and season of the beginning quarter;
the year and season of the ending quarter;
id of the instructor}

The attribute *THE-YEAR-OF-THE-ENDING-QUARTER* is not a part of the key, since this attribute is not needed for identification of the offerings. For given beginning quarter and the season of the ending quarter, we can deduce the year of the ending quarter, since we know that the offering is within one academic year.

Example 5-14.

The binary schema of the university application has been converted so far into the schema in Figure 5-6.

PERSON

last-name: String
first-name: String
birth-year: 1870..1990
address: String
id-key: Integer 1:1

STUDENT

INSTRUCTOR

minor
(*m:1*)

major
(*m:1*)

works in
(*m:m*)

the student
(*m:1*)

the instructor
(*m:1*)

DEPARTMENT

name: String 1:m
main-name-key: String 1:1

**COURSE
ENROLLMENT**

final-grade: 0..100
instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
student-id-in-key: Integer

(*m:1*)

**COURSE
OFFERING**

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String

(*m:1*)

COURSE

name-key: String 1:1

the quarter
(*m:1*)

QUARTER

year-in-key: 1980..1995
season-in-key: String

Figure 5-6. The university schema with keys.

5.3. Disjointness of Categories

Step 2. Convert the intersecting abstract categories into disjoint categories by the following procedure for every group of intersecting categories.

> *Example 5-15.*
>
> Figure 5-7 contains a group of three intersecting categories. The categories *STUDENT* and *INSTRUCTOR* share an intersection.
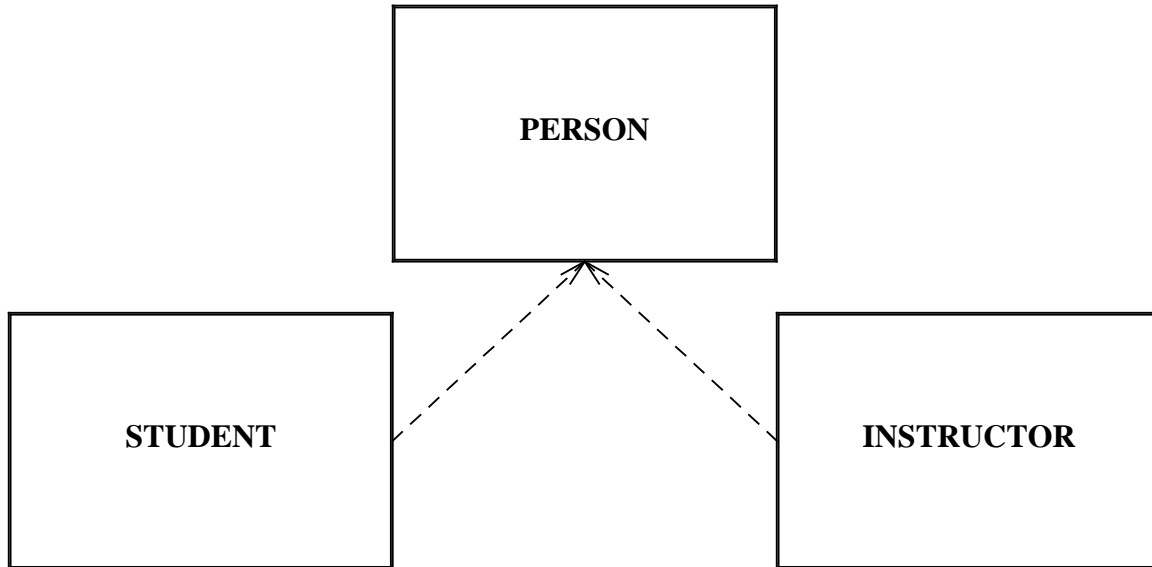
Figure 5-7. Intersecting categories.

A. Consider a **complete group of categories** so that every category outside the group is disjoint from every category in the group.

   Let C denote the union of all the categories in the group. If such a category C does not already exist in the schema, then add it.

   Let $S_1, S_2, \ldots, S_n$ be the other categories in the group. (All of them are direct or indirect subcategories of C.)

> *Example 5-16.*
>
> $C = PERSON$, $S_1 = INSTRUCTOR$, $S_2 = STUDENT$.

Let

$$S_0 = C - \bigcup_{i=1}^{n} S_i$$

S_0 is the hypothetical category consisting of the objects of C which do not belong to any of the subcategories. The category S_0 is considered in order to ensure that no information is lost during the conversion. It is not added to the schema at this time. It may or may not be added to the schema at a later step, depending on decisions made at that step.

Example 5-17.

If there may be other persons in addition to instructors and students, then

$$S_0 = OTHER\text{-}PERSON$$

Otherwise, $S_0 = \varnothing$, and it would not have to be added to the schema at any step.

In the continuation of this case study in the examples we will assume the latter case: $S_0 = \varnothing$, i.e. the database allows no persons other than students and instructors.

B.   Estimate the **intersection factors** ι and ρ.

In order to choose the best way of conversion, we shall need to estimate the following quantities.

Example 5-18.

For the above group of intersecting categories, the choice of the method to eliminate the intersection of the categories will depend on the correlation of two parameters:

- the percentage of people who are both students and instructors, ι, and

- the percentage of relationships specific to students or instructors among all the relationships which can be relevant to persons, ρ.

$$\rho = \frac{number\ of\ relationships\ whose\ domain\ or\ range\ is\ S_1\ or\ \cdots\ or\ S_n}{number\ of\ relationships\ whose\ domain\ or\ range\ is\ C\ or\ S_1\ or\ \cdots\ or\ S_n}$$

Example 5-19.

For the group of the previous examples, $\rho = \dfrac{5}{10}$.

$$\iota = \frac{expected\ total\ number\ of\ objects\ in\ the\ intersections}{expected\ total\ number\ of\ objects\ in\ C}$$

In other words,

$$\iota = \frac{expected\ cardinality\ of\ \bigcup_{i \neq j} (S_i \cap S_j)}{expected\ cardinality\ of\ C}$$

Example 5-20.

To estimate ι, we have to predict the future of our database. It is reasonable to assume that about 5% percent of all persons would be simultaneously students and instructors, so $\iota=0.05$. (Figure 5-8.)
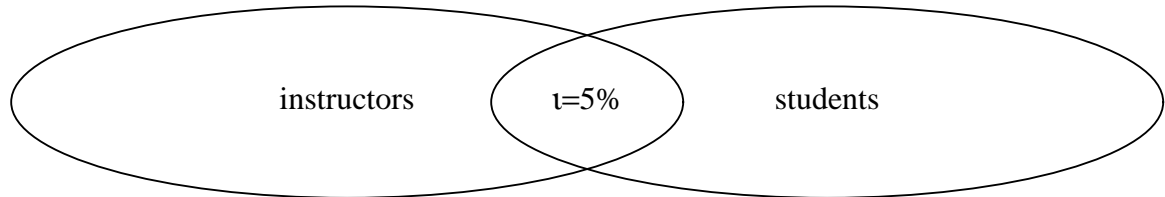
Figure 5-8. An intersection.

Example 5-21.

If we had several intersecting categories, we would count all the intersections (Figure 5-9).
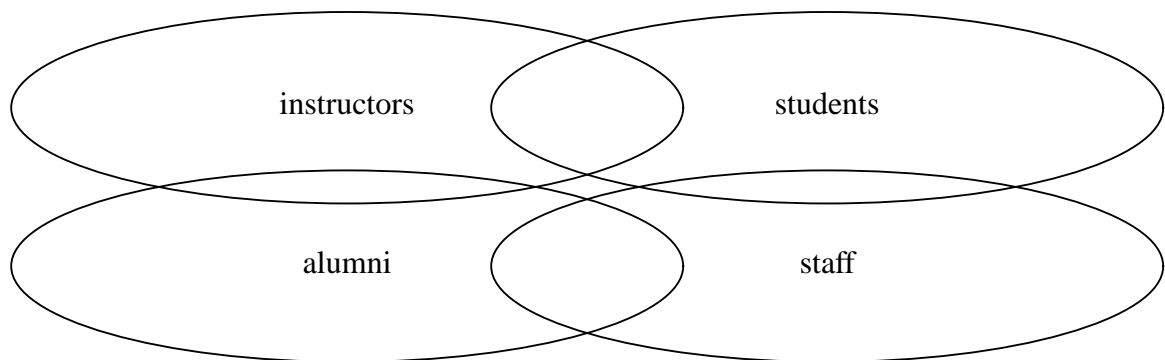
Figure 5-9. Four intersecting categories.

C. Select the best conversion into disjoint categories.

Example 5-22.

Consider the non-disjoint categories *INSTRUCTOR* and *STUDENT*, which are subcategories of the category *PERSON=INSTRUCTOR* $\bigcup$ *STUDENT*. The

following are several possibilities of conversion. We will later select the best of the possibilities, depending on the circumstances.

a. *Conversion into one category (**Union**)*

Example 5-23.

Substitute the whole group of categories by their union, the category *PERSON*. This category will serve as the domain or the range for all the relationships whose domain or range was one of the original categories. In addition, this category will have two Boolean at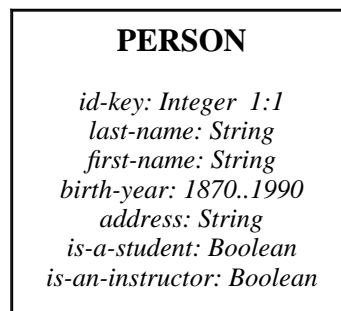tributes, *IS-AN-INSTRUCTOR* and *IS-A-STUDENT*, associating the value *true* with objects representing instructors and students respectively (Figure 5-10).

```
┌──────────────────────────────┐
│          PERSON              │
│                              │
│   id-key: Integer  1:1       │
│   last-name: String          │
│   first-name: String         │
│   birth-year: 1870..1990     │
│   address: String            │
│   is-a-student: Boolean      │
│   is-an-instructor: Boolean  │
└──────────────────────────────┘
```

☐ *major* — relationship from *PERSON* to *DEPARTMENT* (*m:1*)

☐ *minor* — relationship from *PERSON* to *DEPARTMENT* (*m:1*)

☐ *the-instructor* — relationship from *COURSE-OFFERING* to *PERSON* (*m:1*)

☐ *the-student* — relationship from *COURSE-ENROLLMENT* to *PERSON* (*m:1*)

☐ *works-in* — relationship from *PERSON* to *DEPARTMENT* (*m:m*)

Figure 5-10. Category PERSON and its relations.

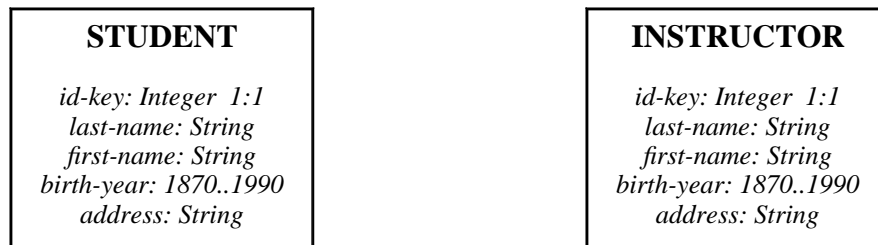b. *Conversion into artificially disjoint categories of **Hats***

Example 5-24.

Substitute these categories by two disjoint categories of hats: *Hat-of-a-STUDENT* and *Hat-of-an-INSTRUCTOR* (usually abbreviated just *STUDENT* and *INSTRUCTOR*, but the meaning of the full names is intended).

An instructor who is also a student will be represented by two distinct

objects of the aforementioned categories.

The objects of the new categories are not persons, but rather their ''hats'' — a person may have two ''hats'', one as an instructor and one as a student. The two categories of ''hats'' are disjoint.

The relationships whose domain or range is the category *PERSON*, for example the relationship *ADDRESS*, will be replaced by two relationships having the new categories as their domains or ranges, such as the relationships *STUDENT'S-ADDRESS* and *INSTRUCTOR'S-ADDRESS*. (Figure 5-11.)
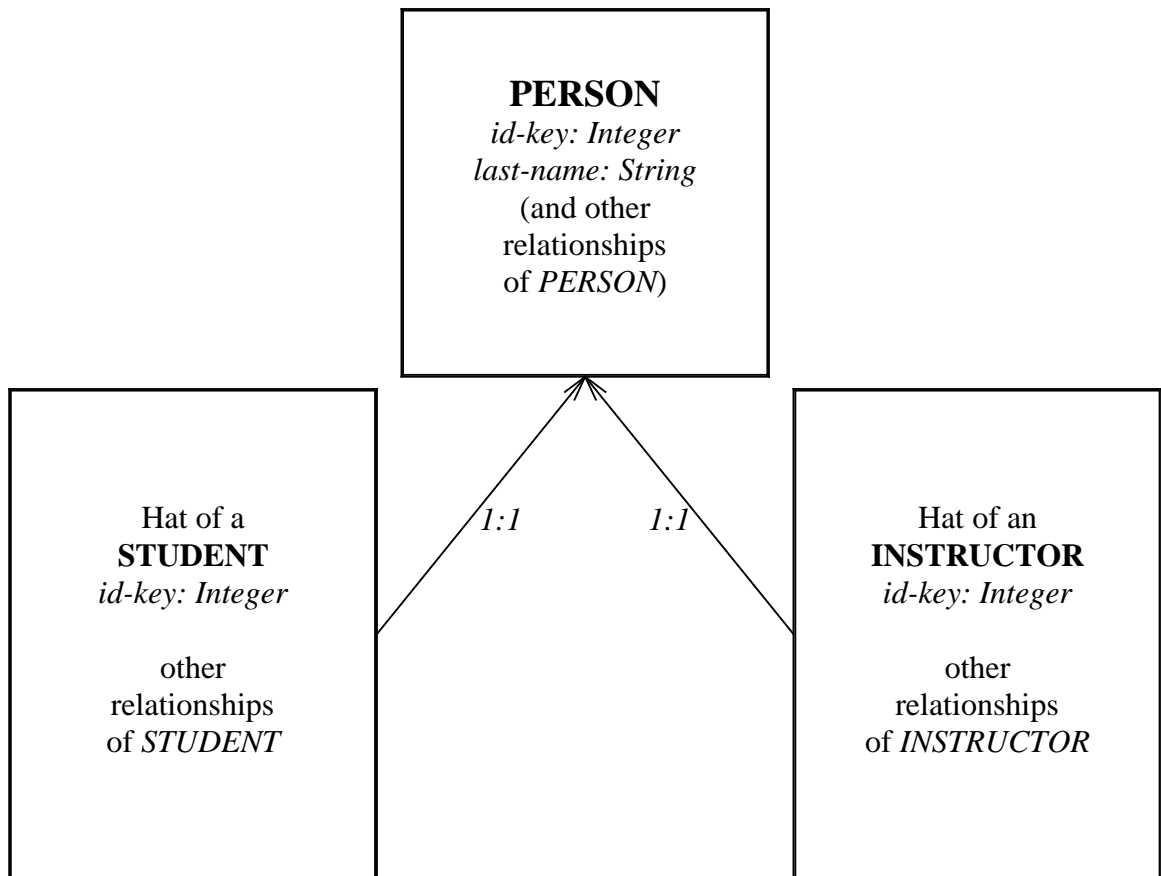
STUDENT	INSTRUCTOR
id-key: Integer 1:1 *last-name: String* *first-name: String* *birth-year: 1870..1990* *address: String*	*id-key: Integer 1:1* *last-name: String* *first-name: String* *birth-year: 1870..1990* *address: String*

Figure 5-11. Attributes moved from PERSON to STUDENT and INSTRUCTOR.

It may appear that by introducing the categories of ''hats'' we have succeeded in defeating the system. Actually, we have fooled ourselves. Without understanding the relationship between two hats of one person, the system will not be able to correctly interpret some queries of naive users, and may cause inconsistency in the stored information and other problems:

- When the address of a person is updated, it may get updated in one category, but not in the other. The database will become inconsistent.

- A naive query like "How many people are there?" will involve double count of persons who are instructors and students simultaneously.

c. *Conversion into* **Union+Hats** *: we can retain the union category with its original relations and in addition have the hat categories to hold relations specific to the former subcategories.*

Example 5-25.

We can retain the category *PERSON* with all its relationships and define two categories of hats which will inherit all the relationships of *STUDENT* and *INSTRUCTOR*, and additionally will have keys and special 1:1 relationships with the category *PERSON* (Figure 5-12.)

- □ *PERSON* — category   (retains its relationships from the binary schema)

- □ *Hat-of-a-STUDENT* — category   (inherits all the relationships of *STUDENT*)

- □ *Hat-of-an-INSTRUCTOR* — category   (inherits all the relationships of *INSTRUCTOR*)

- □ *student-person* — relationship from *Hat-of-a-STUDENT* to *PERSON*   (*1:1,total*)

- □ *instructor-person* — relationship from *Hat-of-an-INSTRUCTOR* to *PERSON*   (*1:1,total*)

Figure 5-12. The category PERSON in addition to the categories of *hats*.

Example 5-26.

To further explore the differences between the three approaches, consider the formulation of the query ''Print the names of all the students.''

Hats :

```
        get s.LAST-NAME
            where s is a STUDENT
    Union :
        get s.LAST-NAME
            where
                s is a PERSON and
                s.IS-A-STUDENT
    Union+Hats :
        get p.LAST-NAME
            where
                p is a PERSON and
                exists s in STUDENT:
                    s.ID-key = p.ID-key
```

Relative disadvantages of each approach

The principal disadvantage of **Hats** is the *redundancy*. For example, the birth-year of an instructor who is also a student has to be logically represented twice in the database, which can cause inconsistency and other problems.

The principal disadvantages of **Union** are the *unnaturalness* of the schema and the *under-coverage of integrity constraints*. For example, an additional integrity constraint has to be defined to prevent association of a non-student instructor with *a major department of studies*. Another important deficiency is the *null-values*, causing significant problems in formulation of queries. (We say that ''*p.MAJOR* is **null**'' if the person *p* is not related to any department by the relationship *MAJOR*.)

The principal disadvantages of **Union+Hats** are the *unnaturalness* of the schema and significant difficulties in the formulation of queries and other operations. These difficulties, however, can be overcome by the use of user-views which would conveniently redefine the concepts of the schema. This requires that the DBMS provide a high level support for user-views, including the capability to specify updates through user-views. Most relational DBMS, however, do not provide sufficient support of user-views.

Conclusion

Unless the DBMS provides sufficient support for user-views as discussed above, we have to exclude the **Union+Hats** approach.

Both other approaches, **Union** and **Hats**, would result in low-quality schemas, but the relational database designer has to choose the better of the two.

> *Example 5-27.*
>
> The choice should usually depend on the correlation of two parameters: the percentage of people who are both students and instructors, ι, and the percentage of relationships specific to students or instructors among all the relationships which can be relevant to persons, ρ.

The relative redundancy in **Hats** increases when ι increases and when ρ decreases. The unnaturalness and the undercoverage of constraints in **Union** increase when ι decreases and when ρ increases.

The following provides a decision criterion for an arbitrary group of categories. The decision is made according to the $\iota{:}\rho$ ratio. When this ratio is relatively high then the **Union** approach would usually be preferable.

When there is a chain of sub-sub-categories, the approach **Hats** becomes too complicated, and is not recommended. It is however the most natural approach in the majority of situations, because in the majority of cases ι is small, the subcategory hierarchy is rather flat (no sub-sub-categories), and the DBMS does not provide a sufficient support for user-views.

D. Convert the group of categories into disjoint categories.

   a. **if** the DBMS provides a high level support for user-views, including specification of updates, **then** (**Union+Hats**):

   (i) Substitute every direct or indirect subcategory *S* of *C* in the schema being converted by the category **Hat-of-a**[n]-*S*. Each object in this new category is an event of membership in the category *S*, that is, if *x* is an *S* then "*x* is an *S*" is one element in Hat-of-an-*S*. (The categories of hats are disjoint. For simplicity, the former names *S* may be kept but the new meaning is assumed.)

> *Example 5-28.*
>
> ☐ *[Hat-of-a-]STUDENT* — category
> ☐ *[Hat-of-an-]INSTRUCTOR* — category

> *Example 5-29.*
>
> If we also had
>
> > ☐ *TENURED-FACULTY* — subcategory of *INSTRUCTOR*
>
> then we would convert it into

> ☐ *Hat-of-TENURED-FACULTY* — category

    (ii)   Retain the category C.

> *Example 5-30.*
>
> > ☐ *PERSON* — category

    (iii) Connect every new category of hats S to each immediate supercategory of S by a new relationship. Specify integrity constraints that these new relationships are one-to-one and total.

> *Example 5-31.*
>
> > ☐ *student-person* — relationship from *Hat-of-a-STUDENT* to *PERSON* (*1:1,total*)

    (iv) Let every new category of hats S have all the relationships that the former category S had.

> *Example 5-32.*
>
> > ☐ *major* — relationship from *Hat-of-a-STUDENT* to *DEPARTMENT* (*m:1*)

    (v)   Specify and add a key for every category of hats S. The simplest way to do this is to inherit the key of C.

> *Example 5-33.*
>
> > ☐ *id-key* — attribute of *Hat-of-a-STUDENT*, range: *Integer* (*1:1*)

  b.   ***else if** ι/ρ**is high*** **or*** there is a chain of sub-sub-categories ***then*** (***Union***):

    (i)   Replace the whole group of categories by one category C.

> *Example 5-34.*
>
> > ☐ *PERSON* — category

---

* "High" is relative, a quantitative measure is the designer's decision. In a practical design of databases with over a hundred categories, the author has applied the threshold of $\iota{:}\rho=0.6$ in most instances.

(ii) Bring all the relationships exiting or entering the former subcategories to C.

Example 5-35.

    ☐ *the-student* — relationship from *COURSE-ENROLLMENT* to *PERSON* (*m:1*)

(iii) Add to C total Boolean attributes named **is-a**[n]-S for every direct and indirect subcategory S of C in the schema being converted.

Example 5-36.

    ☐ *is-a-student* — attribute of *PERSON*, range: *Boolean* (*m:1,total*)

    ☐ *is-an-instructor* — attribute of *PERSON*, range: *Boolean* (*m:1,total*)

(iv) Add an integrity constraint stating that any object of C may participate in a former S's corresponding relationship only if the respective function is-an-S gives *true*.

Example 5-37.

for every p **in** *PERSON*:

    if   exists d **in** *DEPARTMENT*:   p *WORKS-IN* d

then p.*IS-AN-INSTRUCTOR*

Example 5-38.

for every p **in** *PERSON*:

    if not (p *MAJOR* **null**)

    then p.*IS-A-STUDENT*

(v) Whenever there are attributes is-an-S_1 and is-an-S_2, where S_1 is a subcategory of S_2 in the original schema, add a constraint enforcing that in terms of the new attributes.

Example 5-39.

If we had:

    ☐ *UNDERGRADUATE* — subcategory of *STUDENT*

then we would add a constraint:

> **for every** s **in** *PERSON*:
>
> > **if** s.*IS-AN-UNDERGRADUATE*
> >
> > > **then** s.*IS-A-STUDENT*

(vi) Whenever there are attributes is-an-S_1 and is-an-S_2, where S_1 and S_2 are disjoint in the original schema, add a constraint enforcing that in terms of the new attributes.

Example 5-40.

If the category *UNDERGRADUATE* was disjoint from the category *INSTRUCTOR*, then we would add a constraint:

> **for every** s **in** *PERSON*:
>
> > **if** s.*IS-AN-UNDERGRADUATE*
> >
> > > **then not** (s.*IS-AN-INSTRUCTOR*)

---

c.   **else**   (**Hats**):

(i) Substitute the categories $S_1, \ldots, S_n$ by the corresponding n categories **Hat-of-a**-$S_1, \ldots,$ **Hat-of-a**-S_n of the events of membership in categories, that is, if x is an S_i then "x is an S_i" is one element in the category Hat-of-an-S_i. (The categories S_i are disjoint. For simplicity, former names S_i may be kept but the new meaning is assumed.)

Example 5-41.

     ☐ *Hat-of-a-STUDENT* — category

disjoint from

     ☐ *Hat-of-an-Instructor* — category

(ii) If there are, or *may* be in the future, objects in *C* that do not belong to any of the subcategories $S_1, \ldots, S_n$, then add a new category S_0 to the schema. This will be the category of the objects that do not belong to any of the subcategories. This category is usually called **other-***C*.

Example 5-42.

     ☐ *OTHER-PERSON* — category

(iii) Replace every relationship *R* whose domain or range is *C*, by *a* relationship of the same name, but having the categories S_i as their domains or ranges. (The relationship *R* is partitioned into several relationships according to the

restricted domains or ranges S_i.)

Example 5-43.

    ☐ *birth-year* — attribute of *Hat-of-a-STUDENT*, range: *1870..1990* (*m:1*)

(iv) Eliminate the category C.

(v) Specify integrity constraints to prevent inconsistency of the redundant information:

> ''If an object x of the category Hat-of-an-S_i has the same key values as an object y of the category Hat-of-an-S_j, then the other relationships of C (inherited by the categories of hats) must be equal for x and y.''

Example 5-44.

We choose this alternative (**Hats**) for the intersecting group of the subcategories of *PERSON* in the case-study database.

    The schema now has redundancy, which should be controlled by an integrity constraint, if possible. The integrity constraint is

for every s **in** *Hat-of-a-STUDENT*:

for every i **in** *Hat-of-an-INSTRUCTOR*:

    if

        (s.*ADDRESS* $\neq$ i.*ADDRESS* **or**

        s.*LAST-NAME* $\neq$ i.*LAST-NAME* **or**

        s.*FIRST-NAME* $\neq$ i.*FIRST-NAME* **or**

        s.*BIRTH-YEAR* $\neq$ i.*BIRTH-YEAR*)

    then s.*ID-key* $\neq$ i.*ID-key*

(Note: The constraint could have been written without negations, ''in a positive spirit'', but then the meaning of the absent values could be misinterpreted.)

Example 5-45.

The binary schema of the university application has been converted so far (by **Hats**) into the schema in Figure 5-13.
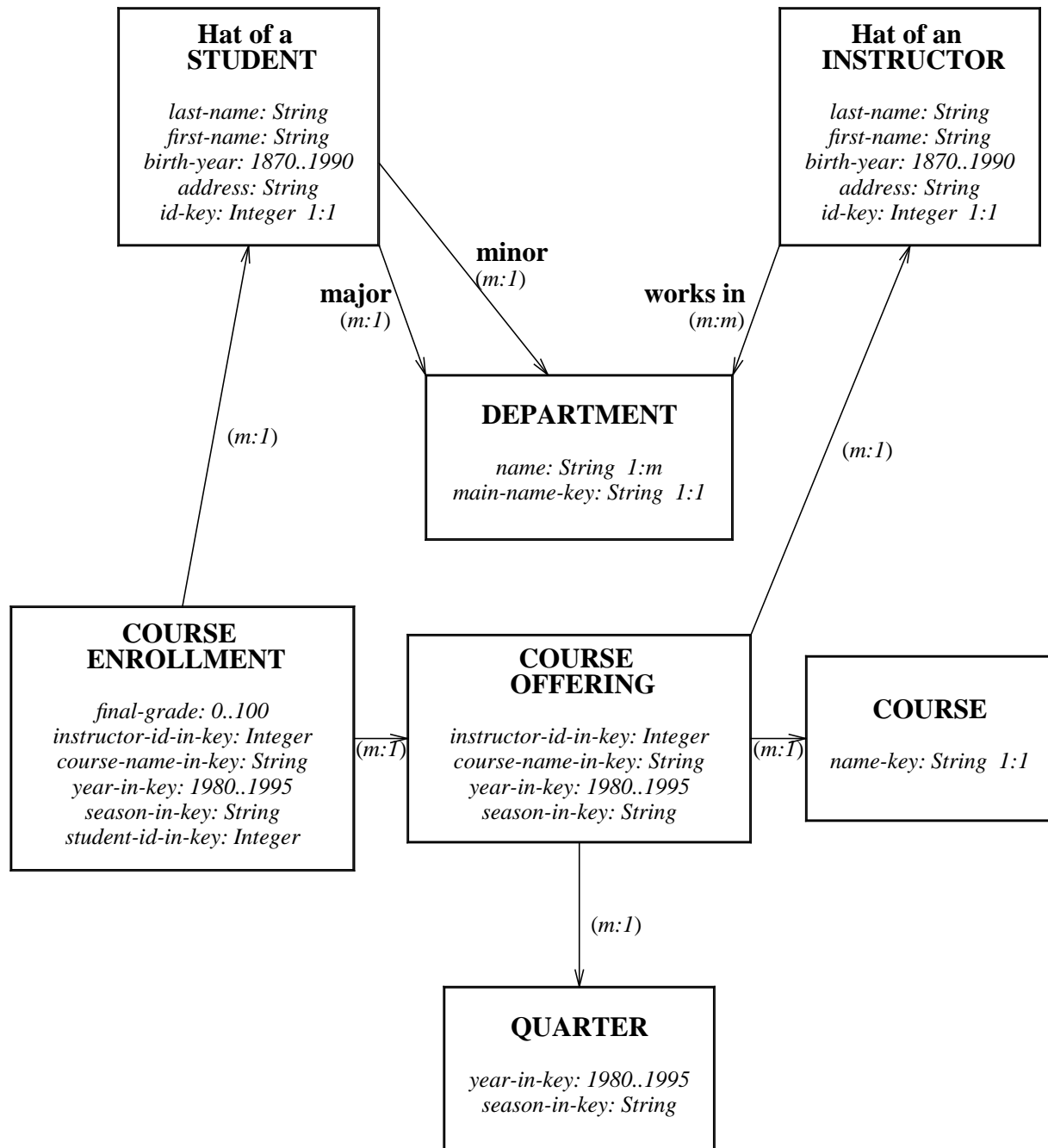
Figure 5-13. The university schema with the categories made artificially disjoint.

5.4. Removal of Relationships

The steps of this section complete the process of schema conversion.

Step 3. Convert every proper **1:m** or **m:m** relationship whose **range is a concrete category** into a new abstract category with its two functional relationships through a relationship-split.

Example 5-46.

Instead of the relationship

  □ *name* — relationship from *DEPARTMENT* to *String*   (*1:m*)

we shall have

  □ *DEPARTMENT-NAME* — category

  □ *the-department* — relationship from *DEPARTMENT-NAME* to *DEPARTMENT*   (*m:1,total*)

  □ *the-name* — relationship from *DEPARTMENT-NAME* to *String* (*1:1,total*)

Step 4. Convert every **1:m** relationship into an m:1 relationship by changing its direction and its name.

Example 5-47.

We do not have such relationships in the university schema. If we assume we have the relationship

  □ *provides* — relationship from *DEPARTMENT* to *COURSE*   (*1:m*)

then we would change it into

  □ *the-department-providing-the-course* — relationship from *COURSE* to *DEPARTMENT*   (*m:1*)

Step 5. Convert every proper **many-to-many** relationship into a category and two functional relationships through a relationship-split.

Example 5-48.

We split the relationship *WORKS-IN* into a new category *WORK* and its two functional relationships *THE-DEPARTMENT* and *THE-INSTRUCTOR*.
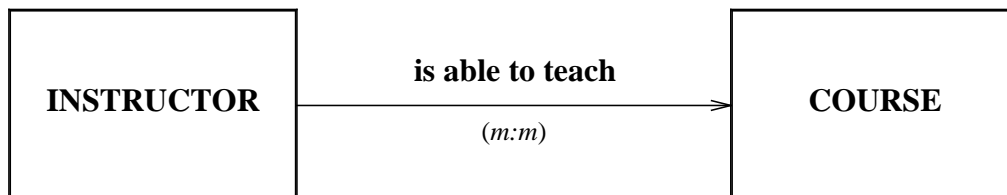
Figure 5-14. A m:m relation

```
                    ┌──────────────┐
                    │  ABILITY TO  │
                    │    TEACH     │
                    └──────────────┘
          the instructor        the course
              (m:1)                (m:1)
    ┌──────────────┐          ┌──────────────┐
    │              │          │              │
    │  INSTRUCTOR  │          │   COURSE     │
    │              │          │              │
    └──────────────┘          └──────────────┘
```
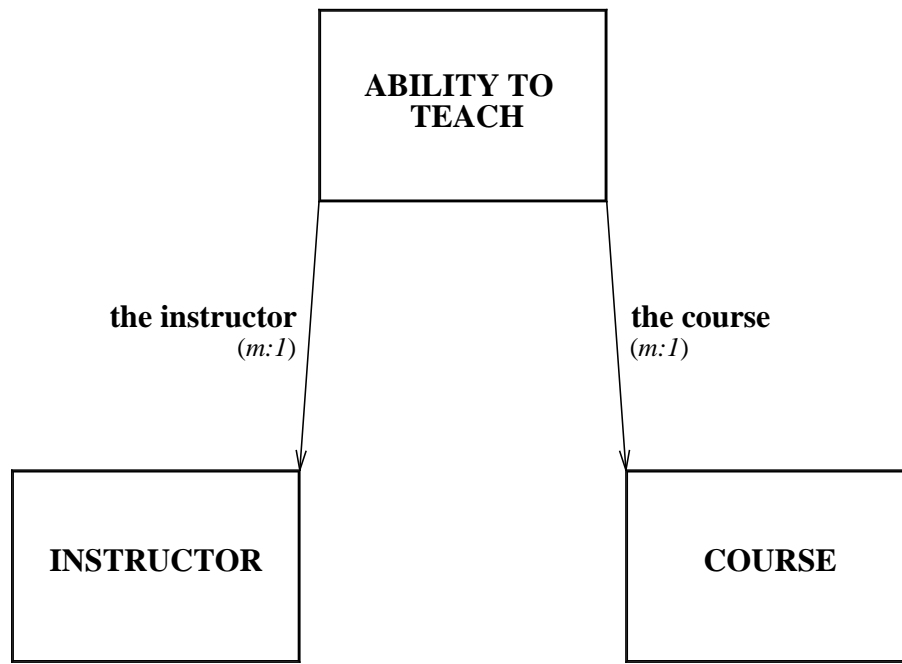
Figure 5-15. Split of a m:m relation.

DEPARTMENT NAME

the-name: String

WORK

STUDENT

last-name: String
first-name: String
birth-year: 1870..1990
address: String
id-key: Integer 1:1

INSTRUCTOR

last-name: String
first-name: String
birth-year: 1870..1990
address: String
id-key: Integer 1:1

(*m:1*)

(*m:1*)   (*m:1*)

minor
(*m:1*)

major
(*m:1*)

(*m:1*)

DEPARTMENT

main-name-key: String 1:1

(*m:1*)

COURSE ENROLLMENT

final-grade: 0..100
instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
student-id-in-key: Integer

COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String

COURSE

name-key: String 1:1

(*m:1*)   (*m:1*)

(*m:1*)

QUARTER

year-in-key: 1980..1995
season-in-key: String

Figure 5-16. The university schema after all the relationship splits have been performed.

Step 6. Choose a key for every category produced through a **relationship-split** as follows.

For every category which was obtained through a relationship-split, a key is contained in the union of the compositions of its two functional relationships on the keys of their ranges.

> *Example 5-51.*
>
> The key of *WORK* is 2 new attributes of this category:
>
> $\qquad$ {main-name of the department, instructor-id of the instructor}

> *Example 5-52.*
>
> The key of *DEPARTMENT-NAME* is contained in
>
> $\qquad$ {the-name, main-name of the department}
>
> Since *the-name* is 1:1, the key of *DEPARTMENT-NAME* is {*the-name*}.

> *Example 5-53.*
>
> The binary schema of the university application has been converted so far into the schema in Figure 5-17.

Step 7. Replace every **m:1 relationship** f whose **range is an abstract category** by the composition of f on the chosen key of its range, that is, by attributes $b_1, \ldots, b_n$, where $x.b_i = (x.f).a_i$, and $a_1, \ldots, a_n$ is the chosen key of f's range.
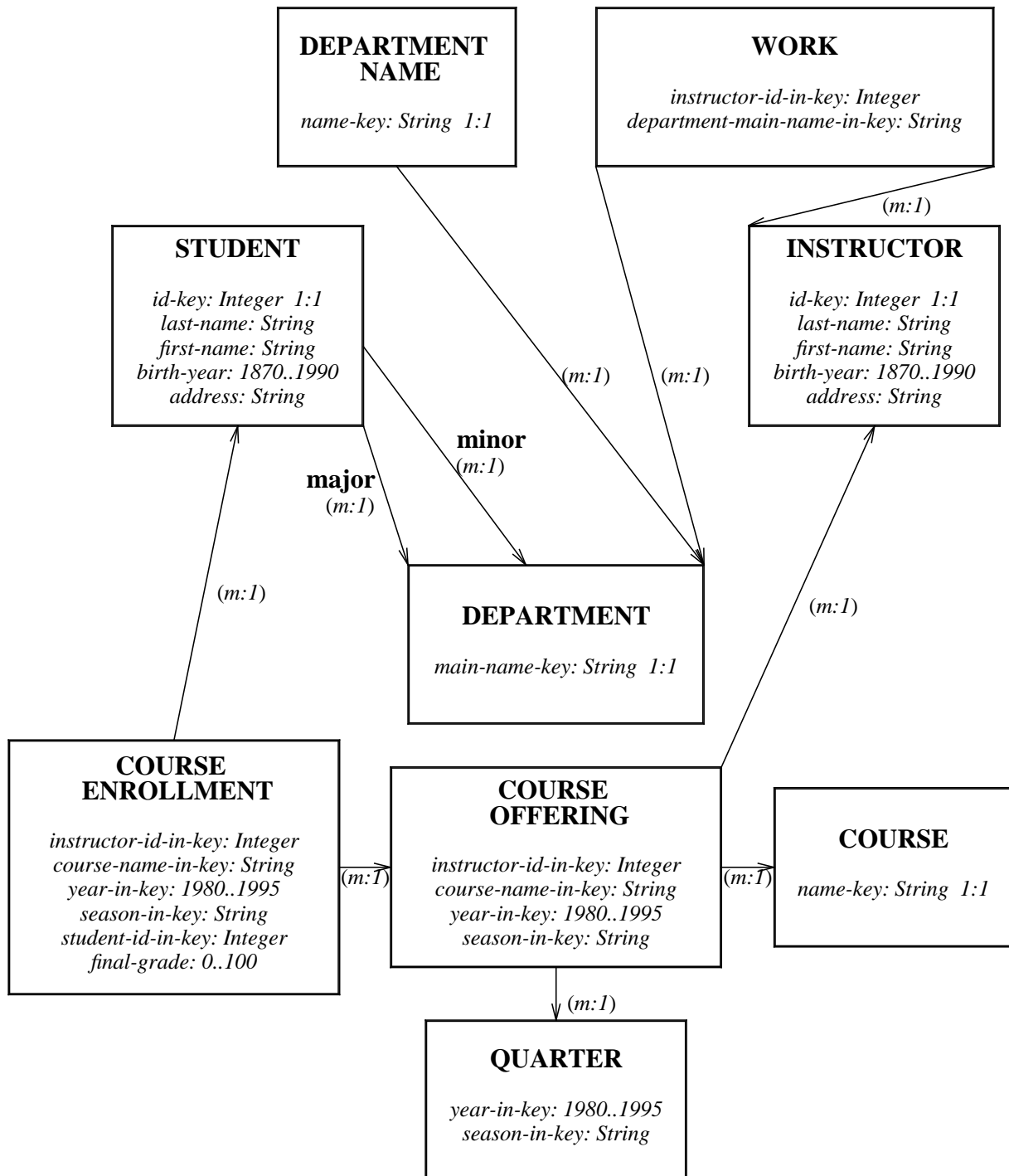
> *Example 5-54.*
>
> Instead of
>
> $\qquad$ ☐ *major* — relationship from *STUDENT* to *DEPARTMENT* (*m:1*)
>
> we shall have
>
> $\qquad$ ☐ *major-dept-main-name* — attribute of *STUDENT*, range: *String* (*m:1*)

DEPARTMENT NAME

name-key: String 1:1

WORK

instructor-id-in-key: Integer
department-main-name-in-key: String

STUDENT

id-key: Integer 1:1
last-name: String
first-name: String
birth-year: 1870..1990
address: String

INSTRUCTOR

id-key: Integer 1:1
last-name: String
first-name: String
birth-year: 1870..1990
address: String

(*m:1*)

minor
(*m:1*)

major
(*m:1*)

(*m:1*) (*m:1*)

(*m:1*)

DEPARTMENT

main-name-key: String 1:1

(*m:1*)

COURSE ENROLLMENT

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
student-id-in-key: Integer
final-grade: 0..100

(*m:1*)

COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String

(*m:1*)

COURSE

name-key: String 1:1

(*m:1*)

QUARTER

year-in-key: 1980..1995
season-in-key: String

Figure 5-17. The university schema after the relationship splits have been performed and keys have been chosen for every category.

Step 8. Remove redundant non-key attributes.

From every category **remove attributes** which are not in the key, but are **inferrable** from other attributes of the same category.

These attributes would usually have resulted from a "blind" application of this algorithm, particularly

a.   A non-key attribute which is always equal to an attribute in the key.

   Note: It is possible that step (7) brought to a category C an attribute b which is always equal to an attribute in the key of C.

b.   A redundant Boolean attribute brought in step (1):

   Suppose

   - We were converting intersecting categories C, $S_1, \ldots,\ S_n$ into disjoint categories,

   - we replaced them by one **Union** category C, and

   - one of the categories S_i was disjoint from all of the rest S_j's.

   Then the new attribute is-an-S_i is inferrable from the rest of the attributes is-an-S_j.

   This attribute should be removed from the schema. (It may be present in a user-view, where it would be an inferred attribute.)

Example 5-55.

If we had

   ☐ *ILLITERATE* — subcategory of *PERSON*   (disjoint from INSTRUCTOR and from STUDENT)

and furthermore, there were no other persons but students, instructors, and illiterate persons, then the attribute *is-illiterate* would be derivable:

   for every p in PERSON:

   p.*IS-ILLITERATE* =

   (not p.*IS-A-STUDENT* and not p. *IS-AN-INSTRUCTOR*)

   Note: The removal of several attributes should *not* be performed simultaneously. Otherwise, two attributes mutually inferrable, but not inferrable from the rest, might be removed.
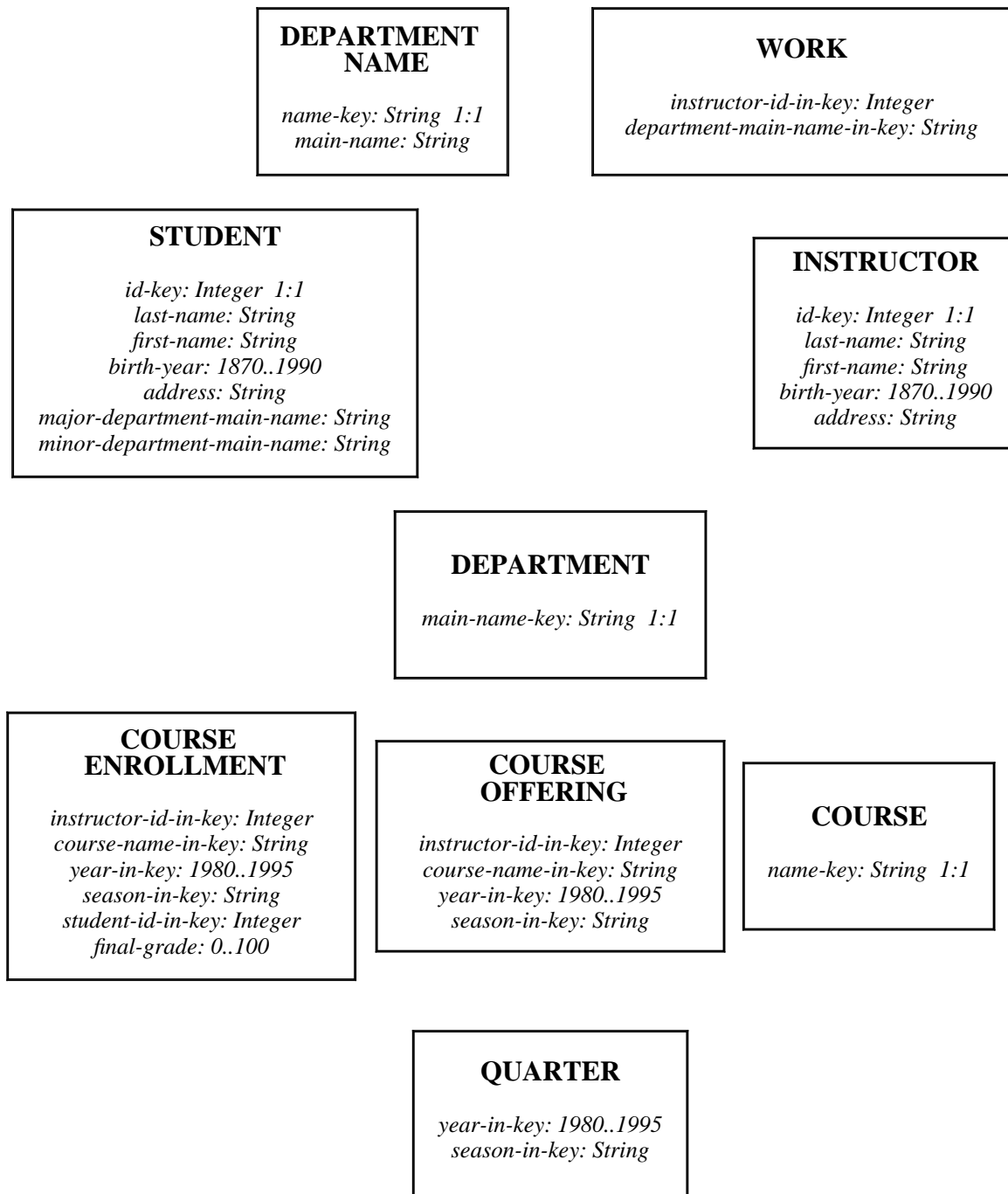
Step 9. Translate the integrity constraints into the terms of the new schema:

a.   The constraints of the original schema.

b.   The additional constraints accumulated during the conversion process.

Example 5-56.

The binary schema of the university application has been converted into the following relational schema.

DEPARTMENT NAME

name-key: String 1:1
main-name: String

WORK

instructor-id-in-key: Integer
department-main-name-in-key: String

STUDENT

id-key: Integer 1:1
last-name: String
first-name: String
birth-year: 1870..1990
address: String
major-department-main-name: String
minor-department-main-name: String

INSTRUCTOR

id-key: Integer 1:1
last-name: String
first-name: String
birth-year: 1870..1990
address: String

DEPARTMENT

main-name-key: String 1:1

COURSE ENROLLMENT

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String
student-id-in-key: Integer
final-grade: 0..100

COURSE OFFERING

instructor-id-in-key: Integer
course-name-in-key: String
year-in-key: 1980..1995
season-in-key: String

COURSE

name-key: String 1:1

QUARTER

year-in-key: 1980..1995
season-in-key: String

Figure 5-18. A relational schema for the university application.

6.   THE TOOL

We have implemented a tool which automates all the busy work of the methodology and provides graphic output. With respect to the intelligent design decisions, the tool accepts instructions from its user, who is a database designer, or, when the user defaults, makes decisions itself based on "rule-of-thumb" principles.

This tool is based on pipeline database design principles: the semantic description of an enterprise is processed by a series of filters, changes in the semantic description are automatically propagated. The input consists of the listing of a linear description of the semantic schema, including the definitions of the meanings of all the categories, relationships, and attributes, integrity constraints at the semantic level, designer's choices for the conversion decisions, and overwrites to be modified in the resulting relational schema. The input consists of sections, each forming a logical subschema. The subschemas are interconnected by common categories. The output of the tool consists of:

1.   Logical design report. This report is independent of the DBMS to be used for the project.

   a.   Graphical semantic subschemas and definitions of all of their concepts.

   b.   Summary of the semantic schema.

   c.   The relational schema and its integrity constraints.

   d.   Glossary, defining the meaning of all the application's attributes and tables.

   e.   Miscellaneous analysis.

   f.   A comprehensive index.

2.   An ORACLE database, including:

   a.   SQL definitions for all the tables, attributes, comments to the attributes (derived from the comments to the meaning in the semantic schema), keys, referential integrity specifications, specifications of checks to be performed on attribute values.

   b.   Generation of screen data entry and update forms, including triggers to enforce integrity. For every table there are two forms: (i) a base form covering all the attributes of the table; (ii) as above, but also containing sub-windows for all the dependent tables connected to this table by 1:m relationships, *i.e.* the tables having referential integrity pointers to this table.

   c.   A generator of default reports.

This tool has been used for database design for the Everglades National Park.

Example 6-1.

A sample logical declaration of an attribute line in the input of the Everglades schema is:

   attr   location-tolerance   HYDROLOGY-STATION   0..1000   m:1
   (Tolerance of the location of a station, in feet. A value X assigned to
   this attribute means that the tolerance is +/-X feet.)

Most of the above is a comment defining the meaning of the attribute. (This comment is automatically propagated to data-entry windows, reports, glossaries, etc.)

The input declarations are maintained in flat files using a text editor. Graphic depictions are automatically generated. Some other approaches prefer graphical input interface. In this tool, we prefer a textual input interface while leaving the pictures for automatic generation by the tool. This allows grater flexibility and saves time. About 80% of the input is the text of the comments that are logical definitions of categories and relations. It is easier to maintain such comment texts using a text editor than using a graphic tool. Also, input hardware independence is achieved: any terminal and a modem will do.

Apart of the design pipeline, the input files are subjected to other tools like spellers, searchers, and publishing systems.

This tool has been implemented at the Florida International University on a SUN-4 computer running a UNIX-compatible operating system. The programs were written mostly in the C language. The database design descriptions are automatically produced in a publication ready form using the DITROFF text processing package. On-screen graphic output is generated in POSTSCRIPT. The current DBMS interface is to the ORACLE system.

7.   THIS METHODOLOGY vs. NORMALIZATION

The *normalization* is a methodology for the design of relational databases has been be quite popular in the academic world. However, it has not been widely used in the application industry. One of the reasons for the lack of popularity in the industry was the mathematical sophistication of the normalization methodology.

This is a ''bottom-up'' methodology. The design proceeds as follows. First, a poor relational schema is designed directly from the requirements. Then, the schema is refined in steps by eliminating certain aspects of redundancy (and thus potential inconsistency and update anomalies). The schema at every step satisfies certain mathematically defined criteria of non-redundancy corresponding to the step. These criteria are called **normal forms** corresponding to the steps.

- The normalization methodology captures only a few of the aspects of the data base semantic quality, while the methodology suggested in this paper attempts to capture all the aspects.

- The normalization methodology is too difficult to be used by most systems analysts and software engineers.

- The normalization methodology is "bottom-up": a "bad" data base is considered to be the starting point, and then it is refined by normalization. This is analogous to writing a bad program and then improving its structure.

This paper's methodology is "top-down": good semantic schemas are designed first, and then they are downgraded to meet implementational restrictions, while the original semantic schemas remain to serve as documentation. This is analogous to writing an algorithm first, and then translating it into a structured program, while the algorithm remains as documentation. In general, this is the advantage of going from conceptual design into logical design. Other approaches exist along the same lines, e.g. [Batini&*al.*-92].

Acknowledgment

References

[Abiteboul&Hull-84] S. Abiteboul and R. Hull. ''IFO: A Formal Semantic Database Model,'' *Proceedings of ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1984.

[Abrial-74] J.R. Abrial, ''Data Semantics,'' in J.W. Klimbie and K.L. Koffeman (eds.), *Data Base Management*. North Holland, 1974.

[Batini&*al.*-92] Batini, C., Ceri, S., Navathe, S.B., *Conceptual Database Design: an Entity Relationship Approach*, Benjamin Cummings, 1992.

[Batory&*al.*-88] D.S. Batory, T.Y. Leung, and T.E. Wise. ''Implementation Concepts for an Extensible Data Model and Data Language.'' *ACM Transactions on Database Systems*, vol. 13, no. 3, September 1988, pp. 231-262.

[Bracchi&*al.*-76] G. Bracchi, P. Paolini, G. Pelagatti. ''Binary Logical Associations in Data Modelings.'' In G.M. Nijssen (ed.), *Modeling in Data Base Management Systems*. IFIP Working Conference on Modeling in DBMSs, 1976.

[Breutman&*al.*-79] B. Breutman, E. Falkenberg, and R. Mauer. ''CSL: A Conceptual Schema Language.'' *Proceedings IFIP TC 2 WG 2.6 Working Conference*, March 1979.

[Brodie&*al.*-84-CM] M.L. Brodie, J. Mylopoulos, and J.W. Schmidt (eds.). *On Conceptual Modelling*. Springer-Verlag, New York, 1984.

[Chan&*al.*-82] A. Chan, S. Danberg, S. Fox, W. Lin, A. Nori, and D.R. Ries. ''Storage and Access Structures to Support a Semantic Data Model.'' *Proceedings of the Eighth International Conference on Very Large Data Bases*. IEEE Computer Society Press, 1982.

[Chen-76] P. Chen. ''The Entity-Relationship Model: Toward a Unified View of Data.'' *ACM Trans. Database Syst. 1*, 1, pp. 9-36.

[DeTroyer&Meersman-86] O. de Troyer and R. Meersman. ''Transforming Conceptual Schema Semantics to Relational Database Applications.'' *Fourth Scandinavian*

Research Seminar on Information Modeling and Data Base Management, 1986.

[Hammer&McLeod-81] M. Hammer and D. McLeod. ''Database Description with SDM: A Semantic Database Model,'' *ACM Transactions on Database Systems*, vol. 6, no. 3, 1981, pp. 351-386.

[Hull&King-87] R. Hull and R. King. ''Semantic Data Models.'' *ACM Computing Surveys, 20*, 3, pp. 153-189.

[Jagannathan&*al.*-88] D. Jagannathan, R.L. Guck, B.L. Fritchman, J.P. Thompson, D.M. Tolbert. ''SIM: A Database System Based on Semantic Model.'' *Proceedings of SIGMOD International Conference on Management of Data*. Chicago, June 1-3, 1988. ACM-Press, 1988.

[Kerschberg&Pacheco-76] L. Kerschberg and J.E.S. Pacheco. ''A Functional Data Base Model.'' Tech. Rep., Pontificia Univ. Catolica do Rio de Janeiro, Brazil, 1976.

[King-84] R. King. ''SEMBASE: A Semantic DBMS.'' *Proceedings of the First Workshop on Expert Database Systems*. Univ. of South Carolina, 1984, pp. 151-171.

[King&McLeod-85] R. King and D. McLeod. ''A Database Design Methodology and Tool for Information Systems.'' *ACM Transactions on Office Information Systems, 3,* 1, pp. 2-21.

[Leung&Nijssen-87] C.M.R. Leung and G.M. Nijssen. ''From a NIAM Conceptual Schema into the Optimal SQL Relational Database Schema.'' *Aust. Comput. J.*, vol. 19, no. 2.

[Mark-83] L. Mark. ''What Is Binary Relationship Approach?'' *Entity-Relationship Approach to Software Engineering*. North-Holland, 1983.

[Mark-87] L. Mark. ''Defining Views in the Binary Relationship Model.'' *Information Systems*, vol. 12, no. 3, 1987.

[Mark-89] L. Mark. ''A Graphical Query Language for the Binary Relationship Model'' *Information Systems*, vol. 14, no. 3, 1989.

[Meersman-81] R. Meersman. ''RIDL: A Query System as Support for Information Analysis.'' *ECODO 32*, September 1981.

[Meersman&Assche-83] R. Meersman and F. Van Assche. ''Modeling and Manipulating Production Data Bases in Terms of Semantic Nets.'' *8th International Joint Conference on Artificial Intelligence, Karlsruhe,* 1983, pp. 325-329.

[Nijssen-77] G.M. Nijssen. ''Current Issues in Conceptual Schema Concepts.'' *Architecture and Models in Data Base Management Systems* North-Holland, 1977.

[Nijssen-81] G.M. Nijssen ''An architecture for knowledge base systems,'' Proceedings, SPOT-2 Conference, Stockholm, 1981.

[Nixon&*al.*-87] B. Nixon, L. Chung, I. Lauzen, A. Borgida, and M. Stanley. ''Implementation of a Compiler for a Semantic Data Model: Experience with Taxis.'' In *Proceedings of the ACM SIGMOD Conference* ACM, 1987.

[Peckham&Maryanski-88] J. Peckham and F. Maryanski. ''Semantic Database Modeling: Survey, Applications, and Research Issues.'' *ACM Computing Surveys, 19*, 3, pp. 201-260.

[Rishe-86-PS] N. Rishe. ''Postconditional Semantics of Data Base Queries.'' *Lecture Notes in Computer Science, vol. 239 (Mathematical Foundations of Programming Semantics.* A. Melton, ed.), Springer-Verlag, 1986, pp. 275-295.

[Rishe-87-RM] N. Rishe. ''On Representation of Medical Knowledge by a Binary Data Model.'' *Journal of Mathematical and Computer Modelling,* vol. 8 (1987), pp. 623-626.

[Rishe-88-DDF] N. Rishe. *Database Design Fundamentals: A Structured Introduction to Databases and a Structured Database Design Methodology.* Prentice-Hall, Englewood Cliffs, NJ, 1988. 436 pp.

[Rishe-88-TM] N. Rishe. ''Transaction-Management System in a Fourth-Generation Language for Semantic Databases.'' In: *Mini and Microcomputers: From Micros to Supercomputers* (Proceedings of the ISMM International Conference on Mini and Microcomputers, Miami Beach, December 14-16, 1988; M.H. Hamza, ed.), Acta Press, 1988, pp. 92-95.

[Rishe-89-SD] N. Rishe. ''Semantic Database Management: From Microcomputers to Massively Parallel Database Machines.'' Keynote Paper, *Proceedings of the Sixth Symposium on Microcomputer and Microprocessor Applications*, Budapest, October 17-19, 1989, pp 1-12.

[Rishe-90-SB] N. Rishe. ''Semantic Binary Database Model.'' *Hong Kong Computer Journal*, 6, *11* (1990), pp. 30-34.

[Rishe-91-PC] N. Rishe and W. Sun. ''A Predicate Calculus Language for Queries and Transactions in Semantic Databases.'' In: *Databases: Theory, Design and Applications.* IEEE Computer Society Press, 1991 (N. Rishe, S. Navathe, and D. Tal, eds.) pp. 204-221.

[Rishe-92-DDS] N. Rishe. *Database Design: The Semantic Modeling Approach*. McGraw-Hill, 1992, 528 pp.

[Rosenthal&Reiner-90] A. Rosenthal and D. Reiner. ''Database Design Tools: Combining Theory, Guesswork, and User Interaction.'' In: *Entity-Relationship Approach to Database Design and Querying*, ed. F.H. Lochovsky. Elsevier Science Publishers (North-Holland), 1990, pp. 187-201.

[Roussopoulos&Mark-85] N. Roussopoulos and L. Mark. ''Schema Manipulation in Self-Describing and Self-Documenting Data Models.'' *International Journal of Computer and Information Sciences*, vol.14, no.1, 1985.

[Senko-78] M.F. Senko. ''Foral LP: Design and Implementation.'' *Proceedings of Very Large Data Base Conference*, 1978.

[Shipman-81] D.W. Shipman. ''The Functional Data Model and the Data Language DAPLEX.'' *ACM Transactions on Database Systems*, vol. 6, no. 1, pp. 140-173, 1981.

[Shoval-85] P. Shoval. ''Essential Information Structure Diagrams and Database Schema Design.'' *Information Systems*, vol. 10, no. 4, 1985.

[Shoval&*al.*-88] P. Shoval, E. Gudes, and M. Goldstein. ''GISD: A Graphical Interactive System for Conceptual Database Design.'' *Information Systems*, vol. 13, no.1, 1988.

[Shoval/Even-Chaime-87] P. Shoval and M. Even-Chaime. ''ADDS: A Systems for Automatic Database Schema Design Based on the Binary-Relationship Model.'' *Data and Knowledge Engineering 2*, 1987.

[Su-88-OS] S. Su. ''An Object-Oriented Semantic Association Model (OSAM*) for CAD/CAM Databases.'' In: *AI in Industrial Engineering and Manufacturing: Theoretical Issues and Applications* (Soyster and Kashyap, eds.), 1988.

[Su&Lo-80] S.Y.W. Su and D.H. Lo. ''A Semantic Association Model for Conceptual Database Design.'' In: P.P. Chen (ed.), *Entity-Relationship Approach to Systems Analysis and Design*, North-Holland, Amsterdam, 1980, pp. 147-171.

[Teorey&*al.*-86] T.J. Teorey, D. Yang, and J.P. Fry. ''A Logical Design Methodology for Relational Databases using the Extended Entity-Relationship Model.'' *ACM Computing Surveys, 18*, 2, pp. 197-222.

[Tsur&Zaniolo-84] S. Tsur and C. Zaniolo. ''An Implementation of GEM — Supporting a Semantic Data Model on a Relational Backend.'' In: *Proceedings of ACM SIGMOD Intl. Conference on Management of Data, May 1984.*

[Verheijen&VanBekkum-82] G.M.A. Verheijen and J. Van Bekkum. ''NIAM - An Information Analysis Method,'' In: *Information Systems Design Methodologies: A Comparative Review*, T.W. Olle et al., eds., IFIP, North-Holland, 1982.

[Vermeir-83] D. Vermeir. ''Semantic Hierarchies and Abstraction in Conceptual Schemata'' *Information Systems*, vol.8, no.2, 1983.