PHENIX Project - BIKIT / FUNDP
IRSIA/IWONL "Tronc Commun" - Contract 5220

PHENIX Project

DATABASE REVERSE ENGINEERING

FINAL REPORT

Volume I : General concepts and Introduction

Second Version - April 1993

About the PHENIX project

PHENIX is a four-year IRSIA/INWONL "Tronc commun" research project, contract no 5220, that started on February, 1st, 1989, and was completed on March, 31st, 1993.

During the first bienniale, its industrial promoters were BBL, Barco, Bell, BIM, E2S, Glaverbel, Groupe S, METSI, Provinces Reunies, Siemens-Nixdorf, Solvay, Sidmar, Telinfo and Warmoes. The second bienniale was supported by BBL, Barco, BIM, E2S, Glaverbel, Groupe S, METSI, Provinces Reunies, Siemens-Nixdorf, Solvay, Sidmar, Telinfo and Warmoes.

The research has been developed by two laboratories, namely the FUNDP and the BIKIT.

The FUNDP (Facultes Universitaires Notre-Dame de la Paix de Namur) team members were M. Chandelon, M. Joris, B. Mignon, C. Tonneau, prof. J-L Hainaut, prof. F. Bodart.

The BIKIT (Babbage Institute for Knowledge and Information Technology, RUG, Gent) team members were E. Cardon, F. Osaer, P. Verscheure, R. Van Hoe, prof. Vandamme, prof. Vanwormhoudt (with the kind support of P. Tisseghem and J. D'Hayer during the last months).

WARNING (Version I, 1991)

This guide has been built with contributions coming from seven authors. I have tried to coordinate as much as possible the writing, in order to avoid an organization such as that of some collective books that appear as a loose collection of heteregeneous papers on a common theme.

Inevitably however, each section is strongly coloured by the personality of its author. This personality dictates not only the writing style, but also the way to perceive problems and to present them. Therefore, the result is highly multicoloured. This results in some inconsistencies that may be annoying for the reader. However, it can be seen in a positive way as well : the problem is so complex that it will profit from a multiview approach.

The time factor has also been a major actor in the result.

On the one hand, technically producing a document with a homogeneous style, no redundancies nor inconsistencies, and that covers all aspects of the subject requires a considerable amout of work, (known to be more than proportional to the number of contributors). This requirement is probably more than is available in the laboratories !

On the other hand, working out the topic, besides its industrial interest, is tackled as a research entreprise. Therefore, it is naturally an ongoing and never-ending process. The guide reflects the current state of a four year research, two years after its begining. It is still full of problems to be solved, and many questions have still to be asked.

Nevertheless, the authors have the feeling that the material included in this guide, despite its flaws, can both help field practitioners and trigger interest among scientific readers. They have great expectations on the feedback (both positive and negative) that will come from the industrial partners.

Jean-Luc Hainaut

WARNING (Final version, 1993)

Things have evolved in such a way that I have been alone in charge of writing the second version of this methodological manual. This has had some positive and negative consequences.

On one hand, I hope that the global coherence of the material has been somewhat improved.

On the other hand, I have not had the time required to carry out a complete rewriting of the text. Therefore, I kept the sections that I though were still valid. Quite naturally, some further effort would have been necessary to make the text completely satisfying with this respect. This (hopefully) improved and completed manual has kept some traces of the early contributors, a characteristics that the reader should consider as a positive aspect of the result of a team work, at least for the result of a scientific research project.

Much effort has been dedicated to the technology transfer objective of this manual, particularly through many examples and case studies. I am not sure that this goal has been fully achieved. Indeed, some concepts, techniques and reasonings still remain complex and somewhat difficult to master for the reader. There are two reasons for that :

- the reverse engineering problem has proved to be much more complex that database design alone, since it copes with technical and empirical material, and not with well formalized abstract specifications.
- efficiently training practitioners in such a new domain can be considered as a specific project whose difficulty has been underestimated in this project. Many years were necessary to build consistent knowledge in database design, and to make it available to practitioners. Therefore, much work still remain to be done in this new domain.

Practically speaking, the first version has be reworked as follows :

- Chapter 1 : quite new; includes a complete summary of the proposals; derives from reference [HAINAUT,93a].
- Chapter 2 : the model has been enriched with new constructs.
- Chapter 3 : section 3.8 is new. The other sections have been enriched.
- Chapter 4 : almost unchanged.
- Chapter 5 : new structure and new sections.
- Chapter 6 : almost unchanged.
- Chapter 7 : almost unchanged. Section 7.3.2 is new.
- Chapter 8 : almost unchanged.
- Chapter 9 : almost unchanged.

- Chapter 10 : almost unchanged.
- Chapter 11 : much augmented : 11.1 has been replaced by a long introduction; translation rules for three DBMS have been developed.
- Chapter 12 : replaced by a more in-depth treatment. Derives partly from reference [HAINAUT,93b].
- Chapter 13 : simplified.
- Chapter 14 : simplified.
- Chapter 15, 16 and 17 : new chapters. Case studies in three DBMS.
- Chapter 18 : Bibliography : much augmented.

These chapters have been organized into two separate volumes for obvious material reasons. Volume 1 covers chapters 1 to 5, and forms the introductory sections of the manual. Volume 2 collects chapters 6 to 18 that form the core of the presentation. It comprises the material specifically dedicated to reverse engineering. A third volume includes some more technical material that could be of interest for some readers. These volumes do not substitute for all the documents that have been produced during the project.

Some of the material that has been included in this version is not actually to be considered as strict results of the PHENIX project. In particular, the software process modeling description (Chapters 11 and 12), the case studies (15, 16, 17), and large parts of the technical appendices have been included to give this manual a larger scope and a more complete covering of the technical aspects. In particular, some material has been borrowed from lectures and tutorials I gave in San Francisco [HAINAUT,91b] and in Lausanne (cours postgrade in Database Technology, EPFL, 1993), as well as from various papers presented in CAiSE and ER conferences.

In particular, I have tried to tackle the reverse engineering of CODASYL and Relational databases as well as of COBOL files (Chapters 11, 12, 15, 16, 17). Chapter 12 includes some material related to IMS and TOTAL/IMAGE databases as well.

Jean-Luc Hainaut

FOREWORD

This PHENIX report is dedicated to methodological aspects of Database Reverse Engineering. It is a major end-product of the PHENIX project (in addition to the PHENIX expert system). Its objective is to present in a structured way the knowledge that seems to be necessary to understand and to carry out the reverse engineering of complex files and databases.

It has progressively appeared that reverse engineering could not be supported by simple systematic approaches as was the case for information systems design and development, and specially databases. On the contrary, the process has proved, both theoretically and experimentally, to be intrinsically non-linear, non-deterministic, adaptative and multi-strategy. In addition, most concepts and processes database design is based on, must be strongly understood before undertaking reverse engineering, even if no systematic methodology was used when designing the application.

Consequently, this guide is definitely not a cookbook that would allow anybody to reverse engineer any file or database by simply following suggested receipts. Its objective is both more modest, and, we think, more clever. It consists in presenting the basic concepts relevant to the domain, identifying and analysing the specific problems, and proposing solutions for them. We hope that this guide will help its users (a user is more than a mere reader) understand the fundamental problems of database reverse engineering and to make them able to tackle this complex activity with better assets.

Table of Contents

Volume I : General concepts and Introduction

Volume II : Reverse Engineering

Volume III : Technical appendices

Appendix A : TRANSFORMATION TECHNIQUES FOR DATABASE REVERSE ENGINEERING

Appendix B : PHENIX CARE TOOL - The User's View (Version 2.0)

Chapter 2 - AVAILABLE FUNCTIONALITIES IN THE PHENIX CARE TOOL

2.1 STARTING AND MANAGING O WORK SESSION

2.2 PROJECT HANDLING

2.3 DATA OBJECT EXTRACTION PHASE

2.4 MISCELLANEOUS FUNBCTIONS

2.5 SUGGESTION

2.6 METHOD DEFINITION

2.7 MAN-MACHINE INTERFACE

2.8 REPORTING

Appendix : TOOL CUSTOMIZATION

Appendix C : ADDITIONAL TECHNICAL DOCUMENTS

PHENIX SYSTEM : PHYSICAL DESCRIPTION

PHENIX SYSTEM : SCREEN EXAMPLES

PHENIX SYSTEM : TRANSFORMATIONS SPECIFICATION (FUNCTIONAL)

PHENIX SYSTEM : LIST OF THE TRANSFORMATIONS (TECHNICAL)

PHENIX SYSTEM : OBJECT-BASE CONCEPTUAL SPECIFICATION (Version 2)

PHENIX SYSTEM : NAME PROCESSING (TECHNICAL)

PHENIX SYSTEM : SPECIFICATION OF REAL, THE IMPORT/EXPORT LANGUAGE

PHENIX SYSTEM : SPECIFICATION OF THE INTEGRATION PROCESS

PHENIX SYSTEM : OBJECT-BASE Version 2.03, TECHNICAL SPECIFICATION

PHENIX SYSTEM : OBJECT-BASE OBJECT-BASE IMPLEMENTATION

Chapter 2

DATABASE MODELING

The main concepts of multi-level database modeling are defined. These concepts are organized according to the three level architecture : conceptual, logical and physical levels. The modeling needs of each level are analyzed through specific specification formalisms. A variant of the Entity-Relationship model is proposed as a supporting formalism for conceptual and logical data structures description.

2.1 INTRODUCTION

2.1.1 Data and data models

Data are representations of real-world objects and facts. Due to this role, their structure can be as complex as the organization of the real-world itself.

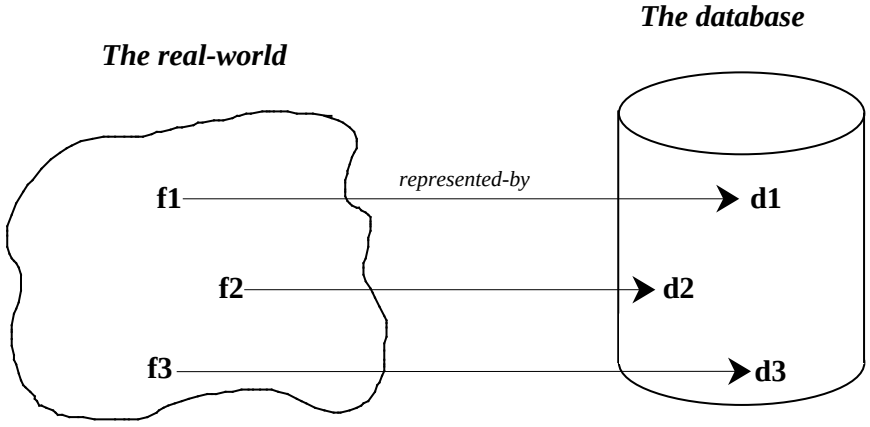


Fig. 2.1 *- Data items d1, d2 and d3 represent real-world facts f1, f2 and f3.*

In order to cope with this complexity, several models of data organization have been developed in the last three decades. Some of these models are closely related to operational products (such as file and database management systems) while others are more general and implementation-independent. These models generally propose a small set of concepts (e.g. entities, record, field), rules stating how to assemble these concepts (e.g. a record is made up of field values), and operators that can be carried out on the concepts (e.g. insert, delete, find). Standard files, IMS, CODASYL, SQL are some examples of operational data models. Entity-Relationship, NIAM and functional models are some examples of implementation-independent models.

2.1.2 Data types and data instances

In the data management realm, it is a tradition to distinguish clearly **data types** and **data instances**[1]. For instance, CUSTOMER_NAME is the name of a data type while 'Smith' is an instance (or an occurrence, or a value) of this data type.

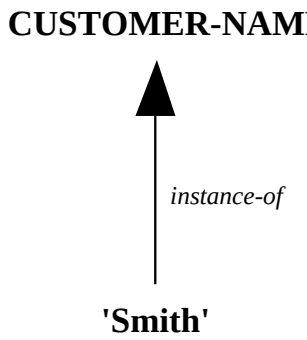


Figure 2.2 *- Data item 'Smith' is an instance of data type CUSTOMER-NAME.*

The time-varying collection of data that represents a real-world domain (from now on, we will call it the application domain as well) is called a database. At a given time, the database is made up of two parts :

1. the **data** themselves, which are an **instance** of the database,
2. the **data types**, which constitute the **schema** of the database.

---

[1] In more advanced knowledge representation models, these worlds are not so insulated from each others. Types (or classes) and instances are objects that can be manipulated with similar functions. Moreover, in some models, the class and instance status are roles that can be played by the same objects.

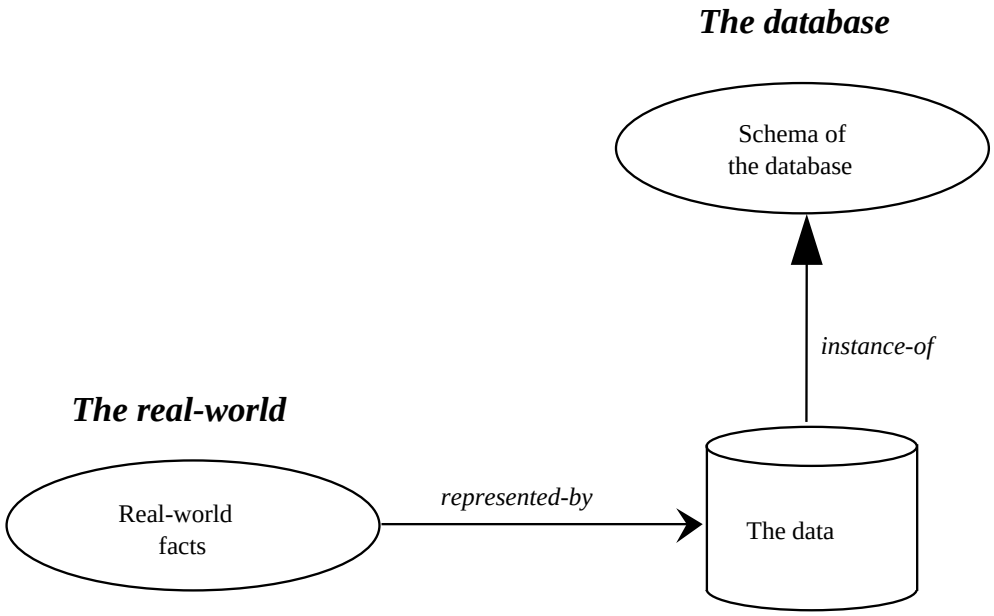


Figure 2.3 *- The database comprises the* ***data*** *(i.e. representation of facts of the real-world) and their* ***schema*** *(i.e. their data types).*

2.1.3 The schemas of a database

The schema governs the way the data are organized and behave. The schema is built according to the concepts and rules of a given data model.

For communication purposes, a schema will be expressed in ad hoc languages, generally referred to as **Data Description Languages** or **DDL**. Some languages are purely textual while others are graphical. Therefore, the same schema can be given several equivalent expressions in different languages.

More than one schema can be associated with a given database. This multiplicity follows two directions :

1. several users (persons or programs) may want to be given different views of these data. Some users want to view subsets of the data or derived data (cf. SQL views, IMS PCB, CODASYL subschemas);
2. the same database can be viewed according to different levels of abstraction. Therefore, a schema can give a very technical view of the data while another schema gives a view that is limited to the logical organization, hiding the technical details (cf. the logical DDL schema and the physical DMCL schema in CODASYL DBMS).

2.1.4 Database description or real-world description

A schema is sometimes interpreted as a description of the application domain itself instead of a description of the data. While such a debate is not closed at the present time, even at the scientific level, and is beyond the scope of this manual, some reflections can be sketched on this point :

- if a schema describes the application domain, the operators of the model it is based on must be real-world-oriented and not data-oriented. Indeed, what would be the meaning of the statements : *create CUSTOMER* and *delete SUPPLIER* if the instances in concern are application domain entities, i.e. actual persons ?
- on the other hand, a schema describes data which, in turn, describe real-world facts. It is therefore easy to compose these description relations by stating that the schema is a representation of the real-world structures (Figure 2.4).

2.1.5 First definitions and principles

From now on, we will adopt the following principles and definitions :

- a schema is a (possibly partial) time-independent description of the organization of a time-varying collection of data;
- a schema is built according to a data model;
- the same collection of data can be given more than one schema;
- a schema can be expressed in a Data Definition Language
- data are representations of facts of the application domain;
- a database is made up of a schema and a collection of data satisfying the schema[2];
- a database can be managed by file management systems (as a collection of files) or by a database management system[3]

---

[2] to be more precise this definition denotes a database state, i.e. the database at a given point of time

[3] these software components will be designated by the acronyms FMS and DBMS in the following. We shall use the generic term DMS, for Data Management System, when no distinction is needed.

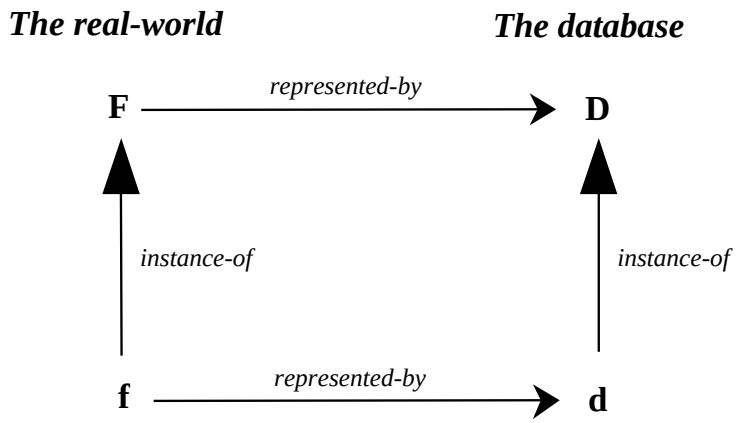


Figure 2.4 *- Fact **f** is of type **F** (i.e. f is an instance of fact type F). Fact **f** is represented by data item **d**. Data item **d** is an instance of data type **D**. Therefore, we can consider that fact type **F** is represented by data type **D**.*

2.2 MULTI-LEVEL DESCRIPTION OF DATABASE STRUCTURES

2.2.1 Levels of abstraction in database description

As already mentioned, the same database can be perceived according to different levels of abstraction, ranging from a very technical description to a high-level, implementation-independent, logical description in which the technical details are hidden. These different schemas are generally dedicated to different actors of the database domain. For instance, an implementation-independent schema can be the enduser's view of the data while the technical schema will be used by database engineers.

In the domain of database design and processing, three levels of description are generally defined, represented by the conceptual schema, the logical schema and the physical schema.

2.2.2 Conceptual description of a database

The conceptual schema defines the aspects of the data structures that represents facts of the application domain. It includes no implementation nor performance-oriented specifications. The model used is made of concepts that can directly model the objects and the structure of the application domain. These models are considered as semantic models, since they represent the meaning aspects of the data related to the world to describe, and (ideally) nothing else. Among most used models we will mention Entity-Relationship models [CHEN,76], Binary models (such as NIAM [VERHEIJEN,82]), functional models [SHIPMAN,81], logic [GALLAIRE,84], semantic networks [WOODS,75], conceptual

graphs [SOWA,84], extended relational models [CODD,79] and object-oriented specification models [BEERI,89].

2.2.3 Logical description of a database

The logical schema is a representation of the data in which operational characteristics are included. The logical schema includes semantic constructs as well, but generally not in a strictly implementation-independent way. It is expressed in terms which are specific to a class of operational DMS. There are few (sometimes none) technical details in a logical schema. Typically, this schema is the view a programmer can have of the database. Among DMS-specific models, let's mention DBTG CODASYL model, IMS model, TOTAL/IMAGE models, SQL models, standard file models and, more recently, object-oriented models. More general models have been defined as well, such as Bachman's Data Structure Diagrams [BACHMAN,69] and the Generalized Access Model [HAINAUT,86]. These models encompass several classes of DMS-specific models, and are used in DB design methodologies.

2.2.4 Physical description of a database

The physical schema is the DMS description of the data. It makes use of concepts specific to the DMS and proposes a technical description of the data. Moreover, it generally defines the way the data are stored and organized on the media, the ways they are retrieved, the performance parameters, etc. This schema is totally DMS-dependent.

2.2.5 Multi-level description in practice

The three-level organization that has been described hereabove is a reference framework. Originated in research and standardization works in the early seventies [DEHENEFFE,74], [ANSI,75], it has influenced DB design methodologies, CASE tools and DBMS architecture. It has been used to analyze and compare specific models as well. However, most DMS do not propose, at least as clearly as stated above, three distinct levels of data description. A DMS model is made up of concepts related to the three levels. Even recent DBMS suffer from this confusion. For instance, some SQL models do not have the conceptual notion of identifier (i.e. relational key). However, a no-duplicates option can be specified as a characteristic of an index, which is a physical concept. In object-oriented models, relationships between objects are defined by including objects into other objects. However, this inclusion defines mainly an access path, and is therefore a logical level concept.

2.3 CONCEPTUAL STRUCTURE MODELING

2.3.1 Conceptual description

A conceptual description of data is aimed at specifying the conceptual structures that underlie these data, that is, their so-called semantics. This description specifies the data as far as they represent application domain objects and facts, independently of any consideration on the computer representation and usage of these data. Since this description is only in terms that are close to real-world concepts, it is called a conceptual description.

2.3.2 The Entity-Relationship model

One of the most widely accepted models for building conceptual descriptions is the Entity-Relationship model. At present time, this model is only used in design methodologies and CASE tools. Though some experiments have been carried out in laboratories, virtually no E-R DBMS have been used for application development so far. Other conceptual models are used in industry, and have led to useful methodologies and design tools (e.g. NIAM [VERHEIJEN,82]). However, none have gained an acceptance that can be compared with the E-R success, despite their qualities. From a scientific point of view, it can be proved that most of these models are formally equivalent, and that they can be derived from a generic model that encompasses all their concepts [HAINAUT,90]. The following presentation is a brief overview of the concepts that will be useful in reverse engineering databases. A more comprehensive specification of this model can be found in [BODART,89] and [HAINAUT,89]. For earlier definitions the reader can consult [DEHENEFFE,74] and [CHEN,76].

The Entity-Relationship model is based on a mental representation of the real-world according to which the world is made up of individual objects, that have properties and that are associated with each other. Accordingly, the data are structured in entities, attributes values and relationships. In addition, the data can be forced to satisfy some restricting rules, called integrity constraints.

2.3.3 Entity-Relationship schema

An E-R schema is the specification of data that describe a part of an application domain[4]. It is composed of the specification of entity types, relationship types, attributes and integrity constraints. More than one E-R schema can be associated with the same application domain. The underlying data can describe different application domain parts,

---

[4] as earlier stated, a common shortcut for this definition can be read an E-R schema is the representation of a part of an application domain.

they can describe the same part, they can also describe the same part with a different point of view.

2.3.4 Entities, entity types and populations

An **entity** is a representation of a major real-world object that is perceived as distinct from all other objects[5].

Similar entities are classified into a specific **entity type**, that is, they are said to belong to a given type. All the entities of a given type share common structural properties : they have values of the same attributes, they can appear in relationships of the same types, they are submitted to the same integrity constraints. The entity types of a schema are given distinct names such as CUSTOMER, PRODUCT, ORDER or SUPPLIER. The expressions "this entity is of (entity) type E" and "this entity is an instance of (entity) type E" will be used alternately.

The **population of an entity type** is the set of entities that belong to this type. This notion is time-varying, since new entities can be created and existing entities can be deleted at any time.

From now on, we shall make use of two notations to express examples. The first notation is text-based while the second one is graphical.

Text notation	Graphical notation
`entity-type CUSTOMER` `entity-type ORDER`	

Figure 2.*5 - Textual and graphical representations of entity types CUSTOMER and ORDER*

2.3.5 Subtypes and supertypes

In some situations, we must accept that a given entity is of more than one entity type. For example, the same entity can be an instance of the types EMPLOYEE, CUSTOMER and SUPPLIER, since the same person can be an employee, a customer and a supplier of the company. The basic construct is called the **is-a relation**. Such a relation can be asserted between an entity type S (called the **subtype**) and another entity type G (called the **supertype**). The interpretation is the following :

---

[5] this definition is basically arbitrary, because it is based on a subjective perception. Therefore, it cannot lead to a unique representation of the real-world. This phenomenon is called semantic relativism, and has induced many research works on proving that two different descriptions are related to the same real-world. This problem will be tackled in the chapters dealing with schema transformation and schema integration.

*at any time, the population of S **is a** subset of the population of G.*

In other words, any instance of S is an instance of G (hence the name *is-a*). For example, if we specify that *WOMAN **is-a** PERSON*, we state that any WOMAN entity is a PERSON entity as well. The *is-a* relation is sometimes called a **Generalization/Specialization** relation, in which S is perceived as a more specific version of G, while G is perceived as a more general version of S. G is called a **generalization** of S, while S is called a **specialization** of G. In the modeling domain, *is-a* relations have some consequences on the attributes, the relationship types and on the constraints of the subtypes that will be studied later (see Inheritance).

An entity type can be given several subtypes :

```
WORKER is-a EMPLOYEE
EXECUTIVE is-a EMPLOYEE
```

These declarations express the fact that workers and executives are subclasses or subsets of the employees.

Conversely, an entity type can be given more than one supertype, i.e. can be a subtype of more than one entity type :

```
OFFICER is-a MILITARY
OFFICER is-a GRADUATE
```

The underlying meaning is that any officer is both a soldier and a graduate. A situation in which an entity type cannot be a subtype of more than one supertype are called single-generalization. Otherwise, we name it multiple-generalization.

The **is-a** relations can be given a graphical representation as follows.

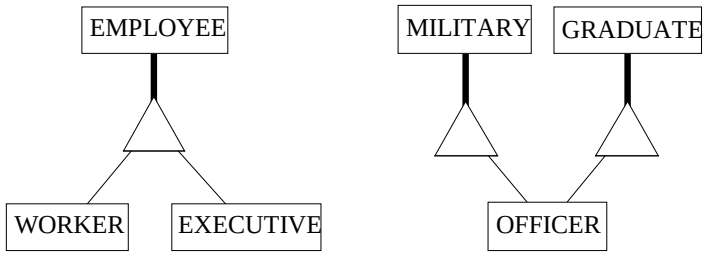
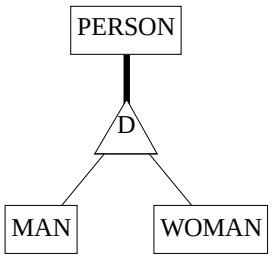


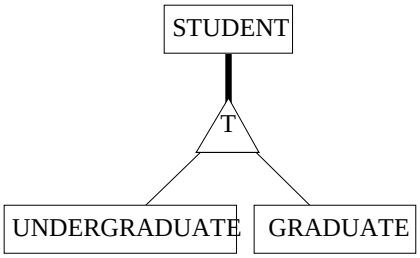
***Figure 2.6** - Any WORKER entity and any EXECUTIVE entity is an EMPLOYEE entity as well. Any OFFICER entity is a MILITARY entity **and** a GRADUATE entity.*

There are two interesting constraints that can be stated on a set of entity types linked with is-a relations, namely the *disjoint* and the *total* constraints.

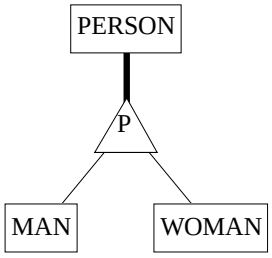
Subsets S1, S2, .., Sn of entity type G are **disjoint** if any entity G may not belong to more than one of these subtypes. It is clear, for example, that MAN and WOMAN are disjoint subtypes of PERSON. We suggest the following graphical convention to state that a person can be a man, a woman, none, but not both (D stands for *disjoint*).



Subsets S1, S2, .., Sn of entity type G are **total** for G if any entity G must belong to at least one of these subtypes. The students who attend lectures in a university can be classified into undergraduates and graduates. Since there are no other categories, we can declare that UNDERGRADUATE and GRADUATE cover STUDENT. We suggest the following graphical convention to state that a student must be undergraduate, graduate, or both (T stands for *total*).



A set of subsets that is both a cover and exclusive is called a **partition**. MAN and WOMAN can be declared a partition of PERSON. However, if a student may be graduated in one department while he/she is still an undergraduate in another one, then UNDERGRADUATE and GRADUATE do not form a partition of STUDENT. We suggest the following graphical convention to state that a person must be a man or a woman but not both (P stands for *partition*).



2.3.6 Relationships and relationship types

Some entities are linked to each other through **relationships**. A relationship (e1,e2,..,en) describes the fact that the real-world objects denoted by the entities e1,e2, ..,en have some relation. In a relationship, each entity plays a specific role. If c is a CUSTOMER entity and p is a PRODUCT entity, the relationship (c,p) states that the customer denoted by c has bought the product denoted by p. In this relationship, c is the BUYER while the role of p is the PURCHASE. The notion of role is particularly important when several entities of a relationship are instances of the same type. If (p1,p2) denotes a parent/child relationship between persons denoted by p1 and p2, it is necessary to specify which entity denotes the parent and which one denotes the child.

Similar relationships are classified into a specific **relationship type**. All the relationships of a given type share common structural properties : they relate the same number of entities, they have the same roles, each role can be played by entities from the same entity type, they have values of the same attributes, they are submitted to the same integrity constraints. The roles of a relationship type are given distinct names. However, if no ambiguity may arise, the name of a role can be the name of the entity type (the default name when the designer doesn't want to give the role a specific name).

Specifying a relationship type, the member entity types and their roles can be as follows,

```
rel-type buys     ( buyer : CUSTOMER,
                    purchase : PRODUCT)

rel-type children ( parent: PERSON,
                    child: PERSON)

rel-type assigned ( order : ORDER,
                    item : PRODUCT
                    assigned-to : SUPPLIER)

rel-type passes   ( customer : CUSTOMER,
                    order : ORDER)
```

Note that the last example, where the role names are the same as the entity type names, can be written in a more concise expression :

```
rel-type passes (CUSTOMER,
                 ORDER)
```

or

```
rel-type passes(CUSTOMER,ORDER)
```

The **population of a relationship type** is the set of relationships that belong to this type. This notion is time-varying, since new relationships can be created and existing relationships can be deleted at any time.

The **degree of a relationship type** is the number of entities each relationship is made up of (i.e. the number of roles). Note that the degree is the number of roles, but by no way the number of entity types. In the *children* example, there is only one entity type, but two roles; therefore, the degree of *children* is 2. Some properties and comments :

- The degree is greater than 1.
- A degree greater than 4 represents complex structures that can be prone to misinterpretation; they will be avoided as far as possible.
- A relationship type with degree 2 is called a **binary** relationship type.
- A schema in which the degree cannot be greater than 2 is called a binary schema.

When an entity type appears more than once in a relationship type, the latter is sometimes called **recursive** or **cyclic**. As already mentioned, specific role names are mandatory in order to avoid ambiguity.

The **cardinality constraint** of role R of relationship type RT is a measure of the number of relationships RT in which any entity can play the role R. The cardinality expresses the *minimum* and *maximum* numbers of relationships in which any entity can, and must, appear. The value *any arbitrary large number*, or *infinity*, is generally represented by **N**. Though any combination of numbers is valid (except N-N, and provided that the minimum value is not greater than the maximum), the most used values are the following :

- 0-1 : at most once;
- 1-1 : exactly once;
- 0-N : any number of times;
- 1-N : at least once.

The examples given above can be completed as follows :

```
rel-type buys     ( buyer [0-N]: CUSTOMER,
                    purchase [0-N]: PRODUCT)

rel-type children ( parent [0-N]: PERSON,
                    child [0-2]: PERSON)

rel-type assigned ( [1-N]: ORDER,
                    item [0-N]: PRODUCT
                    [0-N]: SUPPLIER)

rel-type passes   ( [0-N]: CUSTOMER,
                    [1-1]: ORDER)
```

We shall also make use, when needed, of the graphical notation proposed in Figure 2.7.

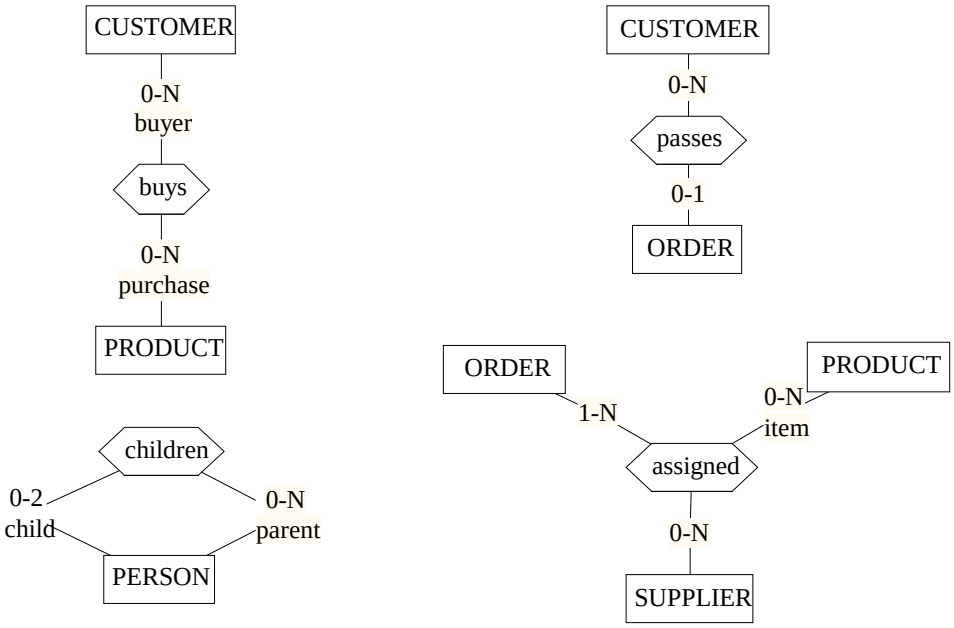


Figure 2.7 *- Graphical representation of four relationship types. Roles are characterized by cardinality constraints (i-j) and by a name. The latter will be specified when it is different from the member entity type.*

A simplified vocabulary is commonly used for binary relationship types (***i*** stands for either 0 or 1):

- **one-to-many** relationship type : one role has cardinality constraint **i-1**;
- **one-to-one** relationship type : both roles have cardinality constraints **i-1**;
- **many-to-many** relationship type : no role has cardinality **i-1**.

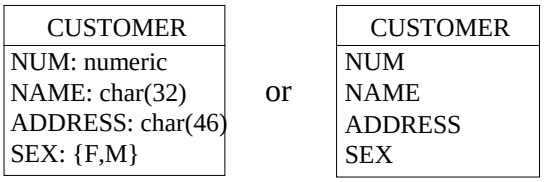
For instance, BUYS is a many-to-many relationship type and PASSES is a one-to-many relationship type.

On the other hand a role with cardinality constraint 0-j is said to be optional (since entities may not play this role) while a role with cardinality constraint i-j, with i > 0, is said to be mandatory (since every entity must play this role). In *assigned* for instance, *ORDER* is mandatory for ORDER while *item* is optional for PRODUCT. When no ambiguity may arise, this property can be declared for the relationship type itself : *assigned* is mandatory for ORDER and optional for PRODUCT.

2.3.7 Attribute values, attributes and domains

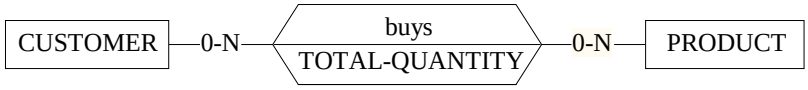
Proper characteristics of a real-world object (represented itself by an entity) are represented by values associated with this entity[6]. For instance, a customer whose name is Adam Smith will be represented by entity e of type CUSTOMER with which the data value "ADAM SMITH" is associated. In general, every CUSTOMER entity will be associated with a similar value. We will say that entity type CUSTOMER has an **attribute** called NAME. "ADAM SMITH" is the **value of attribute** NAME of entity e.

Every attribute draws its values from a specific value set called a **domain**. The domain of NAME of CUSTOMER is, say, the set of all character strings no longer than 32. Domains such as character strings, numbers, logical, real, text, etc., are very basic. Some domains can be specified exhaustively by enumerating all their values. So could be the case for such attributes as SEX, MARITAL-STATUS, STATE, PRODUCT-CATEGORY, etc., where the set of possible values is known and limited. The attributes of an entity type are represented as follows:




```
entity-type CUSTOMER ( NUM : numeric,
                       NAME : char(32),
                        ADDRESS : char(46),
                       SEX : {F,M})
```

A relationship type can be given attributes as well. For instance the *buys* relationship type can be extended with attribute TOTAL-QUANTITY, that precises, for each customer/product association, the total quantity of the product that has been purchased by the customer :




```
rel-type    buys ( buyer[0-N]: CUSTOMER,
                   purchase[0-N]: PRODUCT,
                   TOTAL-QUANTITY : numeric)
```

An attribute of an entity type or of a relationship type can be mandatory or optional, single-valued or multivalued, elementary or compound.

---

6 note that an entity has an existence independent of its attribute values, and is by no means defined as "the concatenation of its attribute values". In particular, an entity type need not have attributes.

Attribute A of entity type E (or relationship type RT) is **mandatory** if a value of A is associated with every entity E. Otherwise, A is **optional**. For instance, attribute NAME of entity type PERSON is mandatory while MAIDEN-NAME is optional.

Attribute A of entity type E (or relationship type RT) is **single-valued** if at most one value of A can be associated with any entity E. If E entities can be given more than one value of A, the attribute is **multivalued**. NAME of PERSON is single-valued while CHRISTIAN-NAME is multivalued.

It is possible to specify both the mandatory/optional and single-valued/multivalued characteristics of an attribute in a way that is similar to the cardinality constraint of a role : the **repeating factor**. This constraint expresses the minimum and maximum numbers of values that can be associated with any entity (or relationship).

Here follows some examples :

- `0-1` : A is optional, single-valued;
- `1-1` : A is mandatory, single-valued (it must be given from 1 to 1 values);
- `0-3, etc` : A is optional, multivalued (it must be given from 0 to 3 values);
- `1-2` : A is mandatory, multivalued (it must be given from 1 to 2 values);
- `5-5` : A is mandatory, multivalued (it must be given exactly 5 values).

The most common combination being 1-1 (mandatory, single-valued), its specification will be omitted.

Attribute A of entity type E (or relationship type RT) is **compound** if A values can be decomposed into meaningful **components**. Otherwise, the attribute is **elementary** or atomic. NAME of PERSON is elementary, while ADDRESS is compound. In the latter case, the components could be NUMBER, STREET and CITY. The components of a compound attribute are still called *attributes*. They are associated with their parent attribute. Therefore, an attribute can be associated with an entity type, a relationship type or a compound attribute. An attribute that is associated with an entity type or a relationship type is called a **non-component** attribute. The **parent** of an attribute is the entity type, relationship type or compound attribute which it is associated with[7].

The text and graphical notations used above can be extended as follows :

```
entity-type PERSON( NUM : numeric,
                    NAME : char(32),
                    CHRISTIAN-NAME[1-4] : char(20),
                    MAIDEN-NAME[0-1] : char(32),
```

[7] to be quite precise, the compound/elementary property should be defined on domains, and should be inherited by the attributes defined on them. However, this would make the model more complex.

```
        ADDRESS (  NUMBER : numeric,
                   STREET : char(35),
                   CITY ( ZIP-CODE[0-1] : numeric,
                          CITY-NAME : char(40))
                )
    )
```

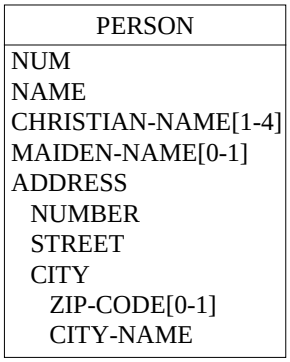


Figure 2.8 *- Representation of the various kinds of attributes (domains omitted)*

2.3.8 Entity type identifiers

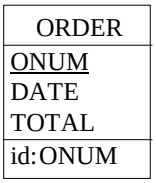
In the simplest cases, an identifier of entity type E is an attribute A such that, in any database instance, there cannot exist more than one entity with a given value of that attribute. Any value of A identifies (i.e. denotes) one entity (or none) among all E entities. Attribute ONUM can be declared an identifier (or an identifying attribute) of entity type ORDER, stating that, in the real-world, there are no two orders with the same order number.

```
entity-type ORDER ( ONUM : integer,
                    DATE : date,
                    TOTAL: numeric
                  )
id(ORDER) : ONUM
```

In simple cases, the identifier can be merely specified by underlining its components :

```
entity-type ORDER ( ONUM : integer,
                    DATE : date,
                    TOTAL: numeric
                  )
```

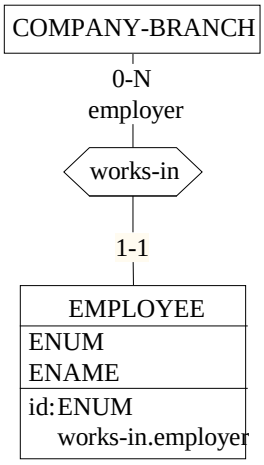
In the graphical notation, both conventions will be used:



An entity type can have more than one identifier. An entity type need not have identifiers[8]. In short, an entity type can have any number of identifiers.

The identifier of an entity type can be made up of several attributes. In this case, there are no two entities with the same values for these attributes.

There are situations where entities can be identified by a combination of attribute values and related entities. Let's consider the following situation : the EMPLOYEE entities linked to a given COMPANY-BRANCH entity by WORKS-IN relationships have distinct ENUM values. Therefore ENUM is a local identifier within WORKS-IN. We can describe this situation by declaring an identifier of EMPLOYEE made up of the attribute ENUM and the entity type COMPANY-BRANCH (via WORKS-IN)[9]. To avoid any ambiguity, the entity component of the identifier is not the COMPANY-BRANCH entity type, but rather its role in WORKS-IN. Through this convention, we can easily take into account situations where there are several relationship types between the entity types or in case of recursive relationship types such as : *a person is identified by his/her father and his/her first name*. The notations are as follows:




```
entity-type COMPANY ( ... )
entity-type EMPLOYEE ( ENUM : ...,
                       ENAME : ..., ... )
rel-type    works-in ( [1-1] : EMPLOYEE,
                       [0-N] : COMPANY )
```

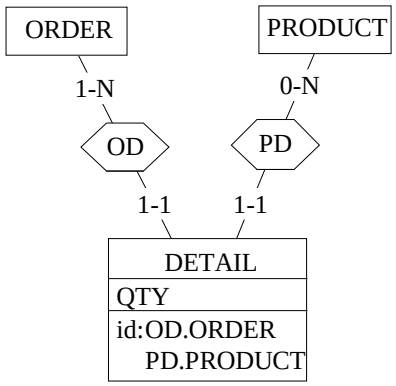
---

[8] this possibility can be useful is some cases, either to represent actual situations or in incomplete schemas. However, it is good practice when designing a database schema to be alerted by such situations, and to check whether no identifiers have been omitted.

[9] this definition is parallel to the corresponding natural language expression : *an employee is identified by its number and the company branch he works in*.

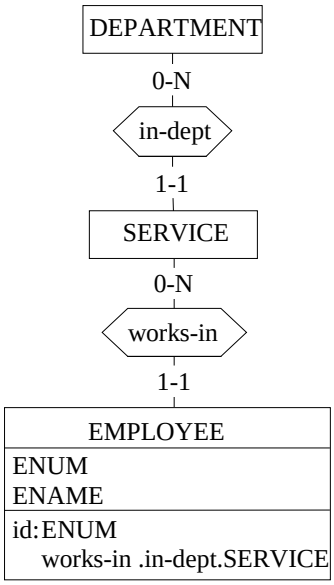
```
id(EMPLOYEE) : works-in.employer, ENUM
```

An identifier can comprise more than one role. This is the case for the DETAIL entity type. If we state that no two lines of order can refer to the same product, then DETAIL entities are identified by their ORDER and their PRODUCT.




```
entity-type ORDER ( ... )
entity-type PRODUCT ( ... )
entity-type DETAIL ( QTY : ...)
rel-type    OD ( [1-1] : LINE,
                 [1-N] : ORDER )
rel-type    PD ( [1-1] : LINE,
                 [0-N] : PRODUCT )
id(DETAIL) : OD.ORDER,PD.PRODUCT
```

Some situations need more sophisticated identifiers, such as in the following example : the employees of a department have distinct employee numbers; on the other hand, the department of an employee is the department of the service which the employee works in. Therefore, an employee is identified by its employee-number and the department of his service.




```
entity-type DEPARTMENT ( ... )
entity-type SERVICE ( ... )
entity-type EMPLOYEE ( ENUMBER : ...)
rel-type    in-dept ( [1-1]: SERVICE,
                      [0-N]: DEPARTMENT )
rel-type    works-in ( [1-1]: EMPLOYEE,
                       [0-N]: SERVICE )
id(EMPLOYEE) : ENUM, works-in.in-dept.DEPARTMENT
```

In this example, the role component belongs to a virtual relationship type obtained by composing the two actual relationship types *works-in* and *in-dept*[10].

Let's mention that we discard identifiers that are composed with only one role. Indeed, this role must have a cardinality i-1. Therefore, this identifier can be considered trivial, and need not be declared. As an example, we can observe that an EMPLOYEE entity alone identifies a COMPANY-BRANCH entity, due to the one-to-many nature of WORKS-IN. This fact is obvious, and there is no need for an explicit identifier stating that EMPLOYEE identifies COMPANY-BRANCH.

A valid identifier of entity type E is made up of one of the following combinations :

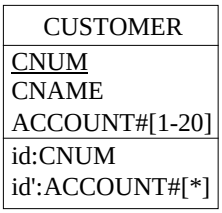
- one or several attributes of E;
- at least two roles with cardinality i-j, with j > 1 and belonging to (actual or virtual) binary relationship types where E plays the other role;
- at least one role with cardinality i-j, with j > 1 and belonging to (actual or virtual) binary relationship types where E plays the other role plus at least one attribute.

[10] given the relationship types R(A,B) and S(B,C), their composition is the relationship type RS such that : (1) if (a,b) belongs to R and (b,c) belongs to S, then (a,c) belongs to RS, (2) if (a,c) belongs to RS, then there exists a B entity b such that (a,b) belongs to R and (b,c) belongs to S.

Two identifiers can overlap, i.e. they can share common attributes and roles. However, an identifier I1 cannot include all the components of another identifier I2. If such is the case, I1 must be discarded since it is only a useless extension of I2. An identifier that doesn't include another identifier is called a minimal identifier. In the situation described above, I1 is called a non-minimal identifier. A good conceptual schema has no non-minimal identifiers.

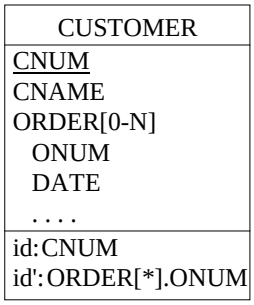
When analyzing physical structures, and particularly unnormalized ones, we will sometimes need even more sophisticated identifying patterns, namely **multivalued identifiers**.

Let's consider the following two examples.




```
entity-type CUSTOMER( CNUM
                      CNAME
                      ACCOUNT#[1-20] )
id(CUSTOMER) : ACCOUNT#[*]
```

There is no two CUSTOMER entities such that their ACCOUNT value sets contain the same value. In other words, an ACCOUNT value identifies at most one CUSTOMER entity.




```
entity-type CUSTOMER( CNUM
                      CNAME,
                      ORDER[0-N] (
                           ONUM,
                           DATE,
                           . . . .)
                    )
id(CUSTOMER) : ORDER[*].ONUM
```

There is no two CUSTOMER entities with the same value of ONUM. Indeed, all orders have distinct order-numbers and an order have been passed by one customer only. Therefore, an ONUM value identifies at most one CUSTOMER entity.

2.3.9 Relationship type identifiers

Identifiers can also be defined on relationship types. Such an identifier is composed of roles and/or attributes. In some cases, the identifier of a relationship type is the set of its roles. For instance, since there cannot be more than one BUYS relationship between the same CUSTOMER and PRODUCT entities, the BUYS relationship type has an identifier composed of the roles buyer and purchase. There are two situations in which an identifier will not be declared :

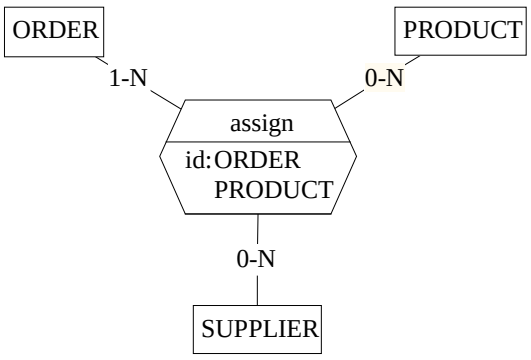
- if role Rj has cardinality 0-1 or 1-1, it is by itself a trivial or implicit identifier of its relationship type (such is the case for PASSES);
- if all the roles constitute an identifier, the latter is considered as a default identifier (such is the case for BUYS).

A valid, non trivial, relationship type identifier is made up of one of the following combinations :

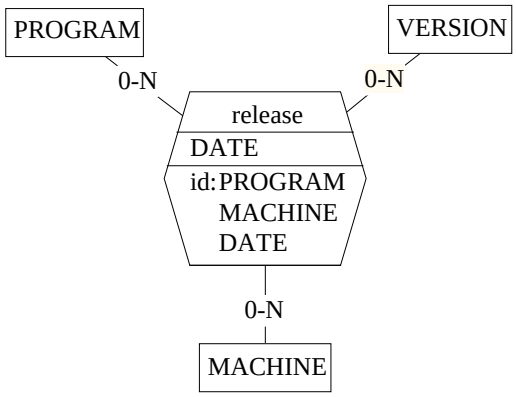
- at least two roles with cardinality i-j, with j > 1;
- one or several attributes;
- at least one role with cardinality i-j, with j > 1 and at least one attribute.

The discussion about the overlapping and the minimality of the identifiers of an entity type is still valid for relationship type identifiers.

The following schema uses the concept of relationship type identifier to state that the product ordered by an order is assigned to one supplier only :



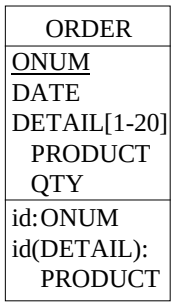

```
rel-type assign( [1-N] : ORDER,
                 [0-N] : PRODUCT,
                 [0-N] : SUPPLIER )
id(ASSIGN) : ORDER, PRODUCT
```



2.3.10 Attribute identifiers

Finally, identifiers can be defined on attributes themselves. Consider an entity type E (the same can be said about relationship types or parent attributes) with a multivalued, compound attribute A, the components of which are A1,A2,..,An. A list of values of A is associated with any entity e of type E. If, in this list of values, there cannot be more than one with a given value of, say, Ai, then we can consider that Ai identifies the A values of any E entity. Ai is said an identifier of attribute A within E.

The following schema illustrate the concept. It is based on the fact that the lines of an order reference different products.




```
entity-type ORDER ( ONUM : ..,
                    DATE : ..,
                    DETAIL [1-20]: ( PROD-NUM: ..,
                                     QUANTITY: .. )
                  )
id(ORDER.DETAIL) : PROD-NUM
```

A valid identifier of the multivalued attribute A is made up of one of the following combinations :

- one or several (direct or indirect[11]) components of A;
- A itself;

Note. The attribute identifier is a special case of normalization structures in non-flat relational schemas. This concept encompasses many situations that can be found in standard file organizations. Moreover, it avoids the introduction of the more general, but much more complex concepts related to (un-)normalized data structures. Indeed the relational theory of normal forms study the normalization state of relational schemas, included non-flat structures[12].

2.3.11 Additional integrity constraints

An integrity constraint is a rule that must be satisfied by any database state[13]. These constraints are an important part of a database schema. The concepts presented so far include several major integrity constraints, such as the identifiers, the cardinality constraints and the repeating factors. Though the list of useful integrity constraints is potentially infinite, we shall describe only some of the most commonly used constraints.

2.3.11.1 Exclusive attributes/roles

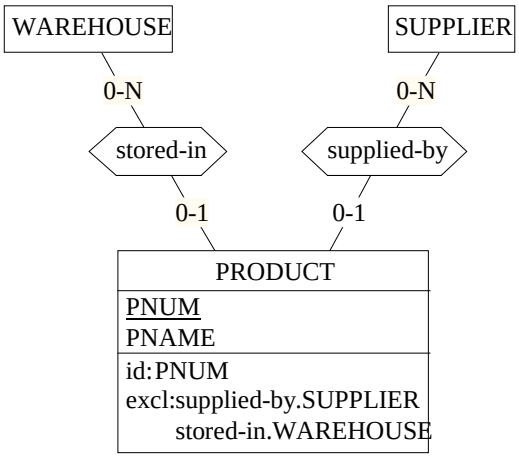
This constraint states that two optional attributes and/or roles cannot be valued simultaneously for the same entity. In the following example, a PRODUCT entity can be linked with a WAREHOUSE entity or with a SUPPLIER entity, but not with both. The roles *stored-in.WAREHOUSE* and *supplied-by.SUPPLIER* are exclusive for PRODUCT :

---

[11] an indirect component of A is a direct or indirect component of a component of A. In the PERSON example of section 2.3.7, STREET is a direct component of ADDRESS while CITY-NAME is an indirect component.

[12] these structures are said to be in *non-first normal form* or N1NF. A N1NF structure can be in turn normalized or unnormalized (see section 9.9. In the latter case, it will imply data redundancies [ULLMAN,88].

[13] this definition is that of static constraints. Dynamic constraints defines valid state transitions, and are not dealt with in this manual.




```
rel-type stored-in(  [0-1]:PRODUCT,
                     [0-N]:WAREHOUSE)
rel-type supplied-by([0-1]:PRODUCT,
                     [0-N]:SUPPLIER)
excl(PRODUCT) : stored-in.WAREHOUSE,supplied-by.SUPPLIER
```

Note that in some cases, this constraint can be expressed with a specialization structure (the example above can be seen as describing two disjoint subtypes of products).

2.3.11.2 Existence dependency

Through this constraint, we state that for each entity e, an attribute has a value, or a role is played by e only when a given condition is satisfied. For instance, the MARRIAGE-DATE attribute of entity type PERSON has a value if and only if the value of the MARITAL-STATUS is "married" or "divorced".

2.3.11.3 Value dependency

This constraint states that the value of an attribute may depend on the values of other attributes. For instance, if the value of LOAN-TERM is less than 1 year, then the value of RATE is 7%, otherwise, it can be given any value.

2.3.11.4 Functional dependency

This a special case of value dependency that will be studied in deeper details in section 9.9. It states that the value of an attribute B always depends on the values of other attributes, say A1, A2, .., An. This property is noted : A1,A2,..,An --> B For instance, the value of LOCATION of entity type EMPLOYEE depends of (i.e. derives strictly from)

the value of DEPART-NAME. This means that all employees working in a given department are located at the unique address of that department.

```
entity-type EMPLOYEE(  ENUM: ..,
                       ENAME: ..,
                       DEPART-NAME: ..,
                       LOCATION: ..)
DEPART-NAME --> LOCATION
```

Due to the uniqueness property of identifiers, there exist, among the attributes of an entity type, a functional dependency from the identifier(s) to any attribute. Therefore, we have also :

```
ENUM --> ENAME
ENUM --> DEPART-NAME
ENUM --> LOCATION
```

However, these constraints are not worth being mentioned explicitly since they derive from the identifier. A similar constraint can be declared for the roles and attributes of a relationship type.

The functional dependencies are an important concept in the relational theory [DATE,86]. It is still valid in other database models, though with a lesser importance [HAINAUT,90b]. A major result from the relational theory applies fully in E-R schemas :

> *if A1,A2,..,An do not constitute an identifier, then the entity type (relationship type) is not normalized, and must be decomposed.*

The phenomenon is easy to illustrate with the example given above, the entities describing the employees that work in the same department have the same DEPART-NAME value, but they also have the same LOCATION value.

This situation violates a basic principle of databases stating that a real-world fact cannot be represented more than once. The single fact that department X is located at address Y is represented as many times as there are employees working in it. This problem will be solved by decomposing the EMPLOYEE entity type as follows (see chapter 9.9) :

```
entity-type EMPLOYEE(  ENUM: ..,
                       ENAME: ..)
entity-type DEPARTMENT(DEPART-NAME: ..,
                       LOCATION: ..)
rel-type WORKS-IN(  [0-N]: DEPARTMENT,
                    [1-1]: EMPLOYEE)
```

In conclusion, a functional dependency that is not originated from an identifier only induces data redundancy. A structure that includes such a dependency is said to be **unnormalized**.

2.3.11.5 Inclusion constraint

According to the relational theory, this constraint states that the values of given attributes must be a subset of the values taken by other attributes. Though this constraint is typical in relational databases and in standard files (i.e. in logical schemas), it can be extended to conceptual schemas as well, specially when we consider the roles of relationship types. Let's consider the following schema, structuring a database that describes orders that refers to products, and suppliers that supply products

```
rel-type REFERS-TO([1-N]: ORDER,
                   [0-N]: PRODUCT)
rel-type SUPPLIES ([0-N]: SUPPLIER,
                   [0-N]: PRODUCT)
```

Observing that an order cannot refer to a product that is not supplied, we can write the following inclusion constraint :

```
REFERS-TO[PRODUCT] is-in SUPPLIES[PRODUCT]
```

where the notation `RT[E]` represents E entities that appear in RT relationships.

The following example is the E-R expression of a so-called referential constraint. This is a special case of inclusion constraint in which the right-side value set is that of an identifier.

```
entity-type ORDER ( ONUM : ...,
                    CNUM : ..., ... )
entity-type CUSTOMER ( CNUMBER : ...,
                       CNAME : ..., ... )
ORDER.CNUM is-in CUSTOMER.CNUMBER
```

2.3.11.6 Redundancy constraints

In principle, a real-world fact must be recorded in the database only once. In operational databases and files, this principle is often violated. In these cases some data have been duplicated. A more subtle situation occurs when derived data are recorded. This is called redundancy.

We have already described an important source of data redundancy, namely unnormalized structures (entity and relationship types). The structural redundancy is another major

source of data duplication. In its most common meaning, this phenomenon is defined by a data item B whose values can always be derived from the values of another data item A.

Practically speaking, should one loose the value of B, it is always possible to compute the lost value from the value of A. The objective is access performance, availability or security. The cost is higher data volume and increased update time. In the following example, CUSTOMER-ADDRESS in ORDER is redundant with ADDRESS in CUSTOMER. This redundancy is expressed with a **copy-of** redundancy constraint.

```
entity-type CUSTOMER( CNUM: ..,
                      ADDRESS: ..)
entity-type ORDER(  ONUM: ..,
                    CUSTOMER-ADDRESS: ..)
rel-type PASSES( [0-N]: CUSTOMER,
                 [1-1]: ORDER)
ORDER.CUSTOMER-ADDRESS copy-of ORDER.PASSES.CUSTOMER.ADDRESS
```

Note that this concept is **not symmetrical**. Indeed, B being redundant with A doesn't mean that A is redundant with B. In the example above, the ADDRESS value in CUSTOMER is not redundant with CUSTOMER-ADDRESS in ORDER, since the former cannot be calculated at any time from the latter.

We cannot state that :

```
ORDER.PASSES.CUSTOMER.ADDRESS copy-of ORDER.CUSTOMER-ADDRESS
```

However, some cases of redundancy are really of a **conceptual nature**. Consider the notion of ORDER. For everybody, the total amount is an integral part of the concept. However, this property is fully derivable when one knows all the information about an order. Therefore, according to the user perception we will add attribute TOTAL-AMOUNT to the ORDER entity type, while the database theorists will disregard this attribute as being a redundant information.

It should be accepted that in some cases, redundant information is kept in a conceptual schema. Should this occur, the expression of the derivation rule (i.e. the redundancy constraint) must be stated, and included in the schema.

2.3.11.7 Coexistence constraint

A set (of at least 2) optional attributes of an entity type can be declared *coexistent*, that is, for each entity, either each of them has a value, or none of them have a value. In the example below, some PERSON entities have both a SPOUSE-NAME and a MARRIAGE-DATE, while the others have no value for these attributes. Anyway, no PERSON entities can have a value for one of these attributes, and no value for the other one.

```
entity-type PERSON(
                    PNUM,
                    NAME,
                    SPOUSE-NAME[0-1],
                    MARRIAGE-DATE[0-1],
                    ADDRESS)

coexist : SPOUSE-NAME,MARRIAGE-DATE
```

The concept can be extended to roles as well. The following example expresses than an employe that works in a department has a HIRE-DATE, while those that do not work has no HIRE-DATE value. Note that the cardinality of WORKS.EMPLOYEE must be 0-j, since this role must be optional. In addition, the *coexist* syntax has been completed to make the object on which the constraint applies explicit.

```
entity-type EMPLOYEE(
                    ENUM,
                    ENAME,
                    HIRE-DATE[0-1])

entity-type DEPARTMENT

rel-type WORKS(     [0-N]: DEPARTMENT,
                    [0-1]: EMPLOYEE)

coexist(EMPLOYEE) : DEPARTMENT,HIRE-DATE
```

The exclusive constraint revisited

The exclusive constraint (2.3.11.1) can be generalized to coexistence groups of components. Indeed, one of the components of an exclusive constraint can be a set of coexistent components (attributes and/or roles) in lieu of an attribute/role. The example below formalizes the fact that if a person works in a company and have a hiring date, then it has no unemployment allowance, and conversely.

```
entity-type PERSON(
                    PNUM,
                    NAME,
                    COMPANY[0-1],
                    HIRE-DATE[0-1],
                    UNEMP-ALLOW[0-1])

coexist : COMPANY,HIRE-DATE
excl : {COMPANY,HIRE-DATE},UNEMP-ALLOW
```

2.3.11.8 Global identifier

An entity type identifier applies on its population, at any time. In some situations, when this population is distributed among two or more entity subtypes, we need to express the fact that there exists an identifier, not only on each of these populations, but also on their union. The following example includes an illustration of such a *global identifier* : foe any value of PID, there may exist either a MAN entity, or a WOMAN entity, or none, but never both of them.

```
entity-type MAN(    MID,
                    M-NAME, ...)

entity-type WOMAN(  WID,
                    W-NAME, ...)

id(MAN + WOMAN): {MID;WID}
```

2.3.12 Inheritance mechanisms

This mechanism is a set of inference rules that apply on the structure of a schema in which is-a relations are defined. Let's consider the declaration S is-a G, stating that entity type S is a subtype of entity type G. Since any S entity is a G entity, it must be given not only values of its own attributes, but also values for attributes of G. In other words, the attributes of S are its own attributes plus the attributes of G.

This rule is called **downward inheritance** of attributes. Let's consider the following schema :

```
entity-type PERSON ( NUM : numeric,
                     NAME : char(32),
                     CHRISTIAN-NAME [1-4] : char(20),
                     ADDRESS : ...)

entity-type WOMAN (  MAIDEN-NAME [0-1] : char(32))

WOMAN is-a PERSON
```

By applying the inheritance mechanism, it can be read as follows :

```
entity-type PERSON ( NUM : numeric,
                     NAME : char(32),
                     CHRISTIAN-NAME [1-4] : char(20),
                     ADDRESS : ...)

entity-type WOMAN (  NUM : numeric,
                     NAME : char(32),
```

```
CHRISTIAN-NAME [1-4] : char(20),
ADDRESS : ...
MAIDEN-NAME [0-1] : char(32))
```

The same can be said for the roles that G plays in relationship types : S is considered as playing these roles as well. If persons can be workers in companies (a fact type that is supposed to be represented by relationships between PERSON and COMPANY entities), so can be women.

When an entity type E is the subtype of several supertypes E1, E2, ..,En (i.e. in case of multiple-generalization), it inherits from all these supertypes, a mechanism called **multiple-inheritance**.

Conflicts may arise when the supertypes have attributes (or roles) with the same names. Such a situation is generally managed as follows :

- either the attributes have different meanings; they are all inherited, but they are renamed in order to give them distinct names;
- or the attributes are the same (e.g. they are themselves inherited from an attribute of a common ancestor); in this case, one only is inherited.

Another inference rule is the **upward inheritance**. According to this mechanism, properties (attributes and roles) of S can be associated with G, where they become optional. In the example above, we can consider that some persons have a maiden name.

Note. Combining both rules leads to a third mechanism, the **sideways inheritance**. Let's consider again the following situation, where UNDERGRADUATE and GRADUATE are not exclusive :

```
GRADUATE is-a STUDENT
UNDERGRADUATE is-a STUDENT
```

If graduates only have diplomas, we are allowed to talk about the diplomas of undergraduates. Indeed, the schema states that some students have diplomas (through upward inheritance); therefore, some undergraduates (who happen to be graduate as well) have diplomas (through downward inheritance). We shall not consider this kind of inheritance so far.

2.4 LOGICAL STRUCTURE MODELING

2.4.1 Introduction

As introduced in 2.2.3, a logical schema defines the data structures of the database according to the model of a DMS, or at least of a class of DMS (e.g. such as relational, CODASYL, hierarchical, standard files). This description includes two kinds of specifications :

- conceptual structures, translated in the concepts that are specific to the DMS data model,
- access structures, that define the ways to access the data.

Therefore, a logical schema is a hybrid description that must be based on a model that can specify both conceptual and technical structures. Moreover, this model must integrate the concepts that can be found in the most commonly used DMS data models.

With that respect, it cannot be one of the available DMS models. Rather, it must provide a small set of general concepts that are both close to the constructs of most DMS data models and independent of each of them. It can be demonstrated that the current DMS models, despite a great variety of vocabulary and of definitions, offer a limited set of very similar structures.

For instance, an IMS segment, a CODASYL record, an SQL row and a COBOL record can be perceived, at a certain level of abstraction, as similar constructs.

Defining a model to express logical data structures allows the specification of DMS-specific schema in a DMS-independent way. It allows also DMS-independent schema manipulation such as schema transformation and integration.

Instead of defining a quite new model, we propose to adapt the Entity-Relationship model to the specific needs of that level of description. The basic concepts of the E-R model are well suited for expressing conceptual structures of logical schemas. The extension to this model will allow the specification of new concepts such as files, access keys (e.g. indices, hash keys, etc), inter-record links (as found in hierarchical and network databases) and ordering (e.g. sorting, FIFO, etc).

2.4.2 Logical interpretation of the E-R concepts

In this logical model, the E-R concepts must be given a slightly different interpretation that can be sketched as follows.

- An **entity type** is the abstraction of such technical constructs as a record type, a segment or a table (the row of which are logical entities). Some logical entities directly describe real-world objects, while others describe more technical constructs, as required by the DMS or by performance considerations. For example, a conceptual entity can be split into two logical entities (= records) to reduce the volume of online files.

- An **attribute** is the abstraction of constructs such as a record (segment) field or a column (SQL).
- A **relationship type** is the abstraction of constructs such as set type (CODASYL), paths (TOTAL/IMAGE) or parent/child links (IMS).
- The concept of **identifier** can be found in all the DMS models, though with some restrictions.

2.4.3 The logical extensions to the E-R model

These extensions describe DMS constructs that are available to the database programmer and that complement the conceptual structure of the schema. These constructs are the file, the access key, the access path and sequence ordering.

2.4.3.1 Files

A file is a dynamic collection of logical entities. According to the DMS, it will be called data file, dataset, physical DB, area/realm, tablespace. In some systems, there can be some restrictions on the types of the entities stored in a file. Some examples of restrictions :

- only one entity type per file (PASCAL files),
- only one file per entity type (COBOL, ORACLE).

Expression :

```
EFILE : collection of EMPLOYEE, SKILL, FOLDER
ESAVE : collection of EMPLOYEE
```

2.4.3.2 Access keys

An access key is an attribute or a list of attributes such that there exists a mechanism that allows a quick, selective, access to the entities that have given values for these attributes. According to the DMS, access keys will be called record/alternate key (COBOL), index (SQL), secondary index, calc key (CODASYL), hash key, inverted file, search key.

An access key has a scope, which is the basic set of entities among which the selection is carried out.

The main scopes are the following :

- all the entities of the database (CODASYL DBkey, relational tid);
- all the entities of a given type (CODASYL 71, SQL index);
- all the entities of a given file (COBOL ISAM);

- all the entities of a given type in a given file (CODASYL 73, IMS);
- all the target entities in a given path (IMS, CODASYL).

Expression :

```
key(PRODUCT) :  PNUMBER
key(CUSTOMER) : NAME
key(EMPLOYEE) : employer, ENUM
```

2.4.3.3 Identifiers and access keys

An important issue is the relation between the identifier and access key concepts. The identifier is basically a conceptual property (integrity constraint) that is independent of any implementation consideration. The access key is an (abstract) technical concept that is aimed at increasing the performance. Theoretically speaking, both are strictly independent. A list of attributes can be an identifier, an access key, both an identifier and an access key, or none.

WARNING

Performance considerations sometimes create some links between these concepts. For instance, enforcing uniqueness of identifier values in update operations can be done efficiently only if the DMS can quickly check this uniqueness in the database. In other words, in case of storing a new entity, the DMS can search the database for the candidate value in a very short time if the identifier is an access key as well. Otherwise, the operation could be very inefficient and time-consuming. Consequently, some DMS impose the restricting condition that any identifier must be an access key. To enforce this rule, the concepts and the syntax to define them offer no way to define an identifier but as a characteristic of an access key (COBOL, CODASYL calc keys, early SQL index). For others, any access key is an identifier and conversely (TOTAL/IMAGE). This unfortunate solution induces much confusion between both concepts[14].

Other DBMS, however, allows the definition of any combination of these concepts. Such is the case in recent versions of SQL (where index declaration is no longer a part of the DDL language) and even in the CODASYL model, where a set search key can, but need not be an index.

[14] to make things worse, the standard relational vocabulary uses the term key to designate identifiers. Shifting from COBOL to relational proves sometimes painful in this respect.

2.4.3.4 Access paths

A binary relationship type RT between entity types E1 and E2 can be perceived by programmers not only as the representation of conceptual relations between E1 and E2, but also as an operational way to get E2 entities from any E1 entity and conversely.

With any relationship type, we can associate **zero**, **one** or **two access paths**. An access path is a mechanism that allows a quick, selective, access to entities playing a role in RT from any entity playing the other role. Therefore, RT can be augmented with two access paths : an access path from E1 to E2 and an access path from E2 to E1.

The starting entity is called the origin of the path while the accessed entities are the targets of the path. Since access paths are implemented with technical constructs (chaining, indices, hash files, inverted lists, etc), they have a cost (space and management time) but they accelerate the execution of some queries. It is therefore useful to distinguish both directions.

According to the current state of the technology, DMS offer binary relationship types only. Thus, an access path is basically a binary mechanism. We can therefore classify access paths as **one-to-many**, **one-to-one**, **many-to-many** and **many-to-one**. The latter case is distinct from the one-to-many case due to the fact that an access path is a directed concept.

The availability of these classes depends on the DBMS :

- CODASYL : one-to-many and many-to-one;
- IMS : one-to-many, some kind of many-to-one;
- TOTAL/IMAGE : one-to-many;
- MDBS III & IV : one-to-many, one-to-one, many-to-many and many-to-one;
- OO-DBMS[15] : one-to-many, one-to-one, many-to-many and many-to-one.

Examples of definition (a path is defined by its origin role):

```
path(PASSES) : CUSTOMER
path(BUYS) : buyer
path(BUYS) : purchase
```

Access path can be defined in logical schemas aimed at DBMS that allow direct representation of relationships, such as network and hierarchical DBMS. Therefore, this concept is not relevant in DMS such as COBOL and SQL systems. In these cases, the main access mechanisms are based on access keys. In fact, there exist schema transformations that replace access paths with access keys and conversely.

---

[15] Object-oriented DBMS.

2.4.3.5 Bag and List attributes

Theoretical information structure models, such the relational model and the Entity-Relationship of NIAM models, have adopted the set-theoretic basic assumption : the population of a type is a *set* of elements. The fact that operation models, such as those of practical DMS are based on a different hypothesis is generally overlooked. Indeed, few (practically none!) include the notion of set of elements, that they can only simulate a set through the concept of identifier.

In particular, the " `occurs N` " of COBOL or CODASYL, and the " `array of` " of PASCAL, and all their equivalents in C, PL/1, BASIC, etc, are not direct expressions of multivalued attributes. Indeed, a multivalued attribute defines *sets of (distinct and unordered) values* associated with each parent object. On the contrary, an array of values in a DMS or in a programming language corresponds to a more complex structure. In such a structure, the values need not be distinct, and there is an order relation on their values (there is a first element, a second one, etc).

In the following example, CAR-MODEL has been declared a **bag** multivalued attribute to express the property that two values of this attributes can be the same for a given CUSTOMER entity, but that their ordering is irrelevant (two cars of a customer can be of the same model). EXPENSE has been declared a **list** attribute to model the fact that two expense values of a given customer can be the same, but that their ordering is significant.

```
entity-type CUSTOMER (  CNUM : numeric,
                        NAME : char(32),
                        CAR-MODEL [0-4] bag: char(20),
                        EXPENSE [0-100] list : numeric)
```

NOTE

Let us observe that, even at the conceptual level, pure multivalued attributes cannot model naturally some common situations. One of the best examples is that of the christian names of a person. Though the christian names of each person are distinct, their position is highly significant : `John Bernard Pickwick` is not the same person as `Bernard John Pickwick`. The christian names can only be modeled through an *ordered set of values*. The example below expresses this fact. Note that an identifier has been defined on the CHRISTIAN-NAME attribute to make the values unique for each PERSON entity.

```
entity-type PERSON (    CNUM : numeric,
                        NAME : char(32),
                        CHRISTIAN-NAME [1-4] list: char(20),
                        ADDRESS : ...)
id(PERSON.CHRISTIAN-NAME) : CHRISTIAN-NAME
```

Unfortunately, no Entity-Relationship (nor any other popular) models include this concept.

2.4.3.6 Sequence ordering

DMS offer several ways to get entities according to a sequence.

Some examples :

- the successive entities of a file;
- the entities of a given type;
- the entities accessed through an access key value;
- the targets of a path.

In some cases, the order according to which the entities are accessed in each of these sequences can be specified. The most common orders are the following :

- `undefined` : the entities are retrieved in an arbitrary order;
- `FIFO`;
- `LIFO`;
- `sorted` : a sort key must be specified;
- `manual` : the position of each entity is user-dependent.

2.4.4 DMS-compliant schema

Such a schema is a logical schema that contains only constructs that are allowed by the target DMS.

Let's consider some examples :

- in an **SQL-compliant schema**, there are no relationship types, there are no multivalued attributes, there are no compound attributes, any identifier must be an access key; etc.
- in a **CODASYL-compliant schema**, there are one-to-many relationship types only, etc.
- in a **COBOL-compliant schema**, there is no relationship types, any identifier must be an access key, any access key must have one component only (but this component may be compound), an access key cannot be included (as a child attribute) into another access key, at least one access key must be an identifier as well, the scope of an access key is all the entities of a file, etc.

These examples suggest that an **X-compliant** schema is a logical schema that satisfies all the characteristics of DMS **X**. Therefore, it is possible to specify a DMS-specific model by a set of rules that define a restriction of an abstract logical model.

2.5 PHYSICAL STRUCTURE MODELING

Despite some early attempts [SENKO,73], there are almost no general models for describing physical structures of files and data bases. At this level, each DMS uses its own set of concepts, its own technical features and its own languages.

A physical schema of a database can be defined as a collection of specifications including DMS texts, parameters, hardware device specifications, etc.

For instance, the physical schema of a collection of COBOL files is made up, among others, of COBOL text fragments from the environment division and the data division.

2.6 A COMMON MODEL FOR DATABASE STRUCTURES

2.6.1 Introduction

In the context of database reverse engineering, we propose a unique model that is to cover most of the data structure specification needs.

Except in steps working directly on the physical description of data, all the data structures will be described through an extended Entity-Relationship model. As suggested in the previous sections, this model will include classical E-R concepts plus logical access concepts. According to the level of abstraction of a description, some concepts will be disregarded, while others will be made visible.

The availability of all the concepts, whatever their abstraction level, allows for very flexible strategies in conducting reverse engineering activities. However, this flexibility can be reduced easily (for instance when the process is controlled by an expert-system) by hiding the concepts that are not relevant in specific steps of the process.

2.6.2 The concepts of the extended E-R model

The concepts are listed hereabove are the building blocks of any database schema. Their definition and their interpretation at the different levels of abstraction can be found in the previous sections.

- Schema
- Entity and entity type
- Subtype and supertype (single and multiple generalization)
- Downward and upward inheritance
- Relationship and relationship type (with any degree greater than 1)
- Role and cardinality constraint

- Attribute of entity type and of relationship type
- Attribute domain
- Compound and elementary attributes
- Repeating factor of an attribute
- Entity type identifier (attributes and/or roles)
- Relationship type identifier (attribute and/or roles)
- Attribute identifier
- Multiple and overlapping identifiers
- Exclusive attributes/roles
- Inclusion constraint and referential constraint
- Redundancy constraint
- Entity collection (at the physical/logical levels, can be used to denote a file)
- Access key
- Access path
- Sequence ordering

2.6.3 Object properties

Each of the objects listed above has specific properties. However, most of them share some common characteristics which are mentioned herebelow.

- Schemas, Entity types, Relationship types, Roles, Attributes, Entity collections have names (included synonyms).
- Schemas, Entity types, Relationship types, Roles, Attributes, Entity collections can have a semantic interpretation, i.e. a natural language or formal text that specifies which real-world facts are described by the concerned data.
- Schemas, Entity types, Relationship types, Roles, Attributes, Entity collections and access keys can have origins. An origin of an object is a conceptual, logical or physical structure from which the concerned object has been derived. An object can have more than one origin. Any reverse engineered object has at least one origin.

2.6.4 Naming conventions

- Schemas, entity types, relationship types, attributes, roles and entity collections have names.
- Each of them has at least one name.
- An object can have more than one name, in which case these names are synonyms.

- When an object has more than one name, it is a good practice to choose one of them as the preferred or standard name in order to simplify schema manipulation.
- Attribution of names to objects must satisfy the following uniqueness rules :
  - the names of the schemas of an application domain are distinct,
  - the entity types of a schema have distinct names,
  - the relationship types in which a given entity type is a member have distinct names,
  - the entity collections of a schema have distinct names,
  - the attributes with the same parent have distinct names,
  - the roles of a relationship type have distinct names.

Chapter 1

INTRODUCTION

This chapter presents briefly the problem of database reverse engineering in organizations. In addition, it includes an overview of the specific problems and of the techniques that will be developed in this manual.

1.1 Database Reverse Engineering : A FIRST CONTACT

Let us first tackle the domain by presenting an obvious example of what database reverse engineering could be. Figure 1.1 proposes an SQL text (top) defining two tables, their columns and their key constraints. Reverse enginering (RE) this (fragment) database consists, among others, in constructing the conceptual schema (bottom) that represents the underlying semantics of this database structure.

```
create table CUSTOMER (      CNUM .. not null,
                             CNAME ..,
                             CADDRESS ..,
                             primary key (CNUM)
                             )

create table ORDER (         ONUM .. not null,
                             CNUM .. not null,
                             DATE ..,
                             primary key (ONUM),
                             foreign key (CNUM) references CUSTOMER
                             )
```

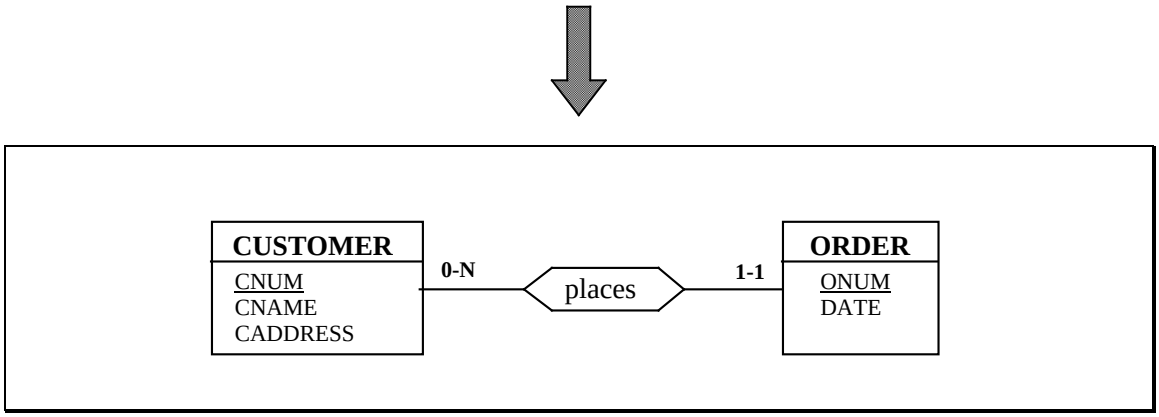


Figure 1.1 *- Database reverse engineering : an idealistic view*

The example is idealistic, and is seldom found in practice. In this example, all the relevant structures are completely expressed in a rich DBMS data language. The designer has chosen explit names and has introduced no complex constructs aimed at optimizing the database. In short, such situations poses practically no problems.

Actual situations are more complex as illustrated in figure 1.2.

```
select CF008 assign to DSK02:P12,
organization is indexed,
record key is K1 of REC-CF008-1.

select PF0S assign to DSK02:P27,
organization is indexed,
record key is K1 of REC-PFOS-1.

fd CF008;
records is REC-CF008-1.
01 REC-CF008-1.
   02 K1 pic 9(6).
   02 filler pic X(125).

fd PF0S;
records are REC-PF0S-1, REC-PF0S-2.
01 REC-PF0S-1.
   02 K1
      03 K11 pic X(9).
      03 filler pic 9(6)
   02 filler pic X(60).
01 REC-PF0S-2.
   02 filler pic X(30).
```

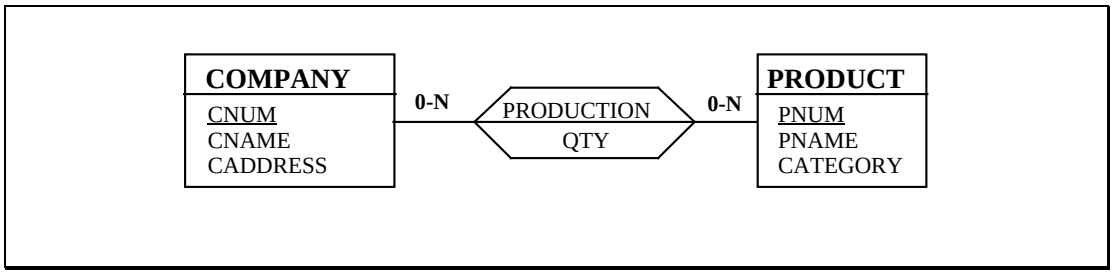


Figure 1.2 *- Database reverse engineering : a realistic view*

This more realistic example, that is far from unusual, exhibits some interesting aspects that makes reverse engineering a difficult task in many cases : names are obscure (though they probably comply with some strict naming conventions), data constraints are absent, field composition is hidden, to name only some problems.

It is obvious that recovering the conceptual schema needs additional informations that could be found in the COBOL programs for example.

Let us suppose that we have found a program that includes the following fragments :

```
01 PRODUCT.
   02 PRO-ID pic X(9).
   02 PADDING pic 9(6).
   02 PNAME pic X(40).
   02 CATEGORY pic X(20).

01 PRODUCTION redefines PRODUCT.
   02 PRO-ID pic X(9).
   02 CNUM pic 9(6).
   02 QTY pic 9(10)V9(4).

...

read PF0S into PRODUCT.
if PADDING = 0 then perform PROCESS-PRODUCT
               else perform PROCESS-PRODUCTION.
```

This text gives us important additional information on the files. For instance :

- the record types are given more explicit names,
- they are given a more precise decomposition,
- the link between the record types can be guessed,
- it seems that further analysis of sections PROCESS-PRODUCT and PROCESS-PRODUCTION should bring a better knowledge on these structures.

Though this example is a bit naive, it illustrates how the analysis of procedural parts of the programs can progressively complement the knowledge provided by the mere program data structures.

1.2 FIRST DEFINITIONS

Roughly speaking, reverse engineering (RE) a piece of software consists in reconstructing its functional and technical documentation, starting mainly from the source text of the programs [IEEE,90] [HALL,92]. Recovering these specifications is generally intended to convert, restructure, maintain or extend old applications. It is also required when developing a Data Administration function.

In short, RE tries to answer the question : *what are possible specifications of this implementation*[1]. The problem is particularly complex with old and ill-designed applications. In this case, not only no decent documentation (if any) can be relied on, but the lack of systematic methodologies for designing and maintaining them have led to tricky and obscure code. Therefore, RE has long been recognized as a complex, painful and prone-to-failure activity, in such a way that it is simply not undertaken most of the time, leaving huge amounts of invaluable knowledge buried in the programs, and therefore definitely lost.

In *data-oriented applications*, the complexity can be broken down by considering that the files or databases can be reverse engineered (almost) independently of the procedural parts.

This proposition to split the problem in this way can be supported by the following arguments :

- the semantic distance between the so-called conceptual specifications[2] and the physical implementation is most often narrower for data than for procedural parts;
- the data are generally the most stable part of applications;
- even in very old applications, the *semantic structures* that underlie the file structures are mainly procedure-independent (though the *physical structure* is highly procedure-dependent);
- reverse engineering the procedural part of an application is generally easier when the semantic structure of the data has been elicited.

Therefore, concentrating on reverse engineering the data components of the application first can be more successful than trying to cope with the whole application. Though RE data structures still is a complex task, it appears that the current state of the art provides us with sufficiently powerful concepts and techniques to make this enterprise more realistic.

---

1 A secondary, but also important question is often *how the implementation got to be what it is*, i.e. elicitating the functional and technical requirements together with the reasonings through which they have been satisfied.

2 In the database realm, the abstract, implementation-independent specification of a database is called its *conceptual schema*.

1.3 A SHORT STATE OF THE ART

Considering the proliferation of comprehensive database design methods and database-oriented CASE tools, database forward engineering appears much more mature than its processing counterpart. This fact is easy to explain. The domain is more restricted and offers less freedom while the requirements are better understood. In addition (and probably consequently) a fairly comprehensive and realistic database theory has now been made available for practitioners, a fact that cannot be claimed for most SE formal approaches.

An increasing number of CASE tools offer some database reverse engineering (DBRE) functionalities (let us mention the Bachman re-engineering toolset only). Though they ignore many of the most difficult aspects of the problem, these tools provide their users with invaluable help to carry out DBRE more effectively [ROCK,90].

Surprisingly enough, DBRE has raised little interest in the DB scientific community. By browsing major information sources such as ACM TODS, VLDB and ER conferences proceedings, or Knowledge & Data Engineering, one can hardly collect twenty papers more or less related with DBRE. Let us mention some references :

- RE of standard files : [CASANOVA,83], [NILLSON,85], [DAVIS,85]
- RE of IMS databases : [NAVATHE,88], [WINANS,90]
- RE of CODASYL databases : [BATINI,92]
- RE of relational databases : [CASANOVA,84], [NAVATHE,88], [DAVIS,88], [SPRING,90], [FONKAM,92]

Most of these studies appear to be limited in scope (most often dedicated to one data model), and are generally based on severely unrealistic assumptions on the quality and completeness of the data structures to reverse engineer. For instance, they suppose that,

- all the conceptual specifications have been translated into data structures and constraints,
- the translation is rather straightforward (no tricky representations),
- the schema has not been deeply restructured for performance objectives or for any other requirements,
- a complete physical schema of the data is available,
- names have been chosen rationally (e.g. a foreign key and the referenced primary key have the same name).

The situation should evolve rapidly , as testified for instance by [BATINI,92], the first popular reference that includes several sections dedicated to DBRE. However, a general theory supporting DBRE of real applications has yet to be built.

1.4 OVERVIEW OF THE MANUAL

This section is derived from reference [HAINAUT,93] that presents in a few pages the principle aspects of database reverse engineering. It can be considered as a summary of this manual

1.4.1 Database design revisited

Though database design has been one of the major theoretical and applied research domain in software engineering in the last two decades, and though it can be considered as a now fairly well mastered problem, we are faced here with files and database[3] that have not been built according to well structured methodologies. Tackling the reverse engineering of a database needs a deep understanding of the forward process, i.e. database design, not only according to standard and well formalized methodologies, but above all when no such rigourous methods have been followed. In other words, we intend to grasp the mental rules that make the intuitive behaviour of practitioners. Our approach should not be **normative**, as in forward engineering, where practitioners are told how to work, but rather **descriptive**, since we have to find out how they have worked.

The analysis is based on the following assumptions about database design :

- a database must satisfy a limited set of **requirements** such as : correctness, user acceptance, time performance, space performance, availability, reliability, compliance with a data manager, hardware constraints, security, compatibility with organizational constraints, conformity with a methodological standard, etc. Satisfying each kind of requirements constitutes an identifiable **design problem**.
- solving each of these design problems is the objective of an identifiable **design process**; each process produces **design products**, i.e. specifications of the solution at a given level of abstraction;
- designing any database, whatever the methodology (or absence thereof), requires to solve the same collection of problems; therefore, designing a database consists in carrying out a **limited set of processes**; in some unstructured design approaches, some processes can be by-passed while in others several processes are conducted in parallel.
- each design process is based on specific **concepts**, **techniques** and **reasonings**;
- each design process can be expressed as a **transformation** applied on input products and yielding output products; the transformation consists in adding constructs to the input specifications, removing some constructs, or changing the format of these specifications.

---

[3] From now on, the term *database* will encompass any permanent, structured, data collection on secondary storage, including standard files.

These assumptions are the basis for a *generic model of database design activities* that gives us some useful hints on how to conduct the reverse engineering of an existing database. Indeed, it suggests to **search the description of the database for traces of each specific design process**, instead of trying to rebuild the conceptual schema in a single step. In addition, it allows for the specification of both non standard and empirical design practices.

In the limited scope of this paper, we shall consider a simplified generic model of database design. Figure 1.3 depicts the organization of the main design processes and the design products. According to this model, the processes that can be carried out and the products they transform and produce are as follows :

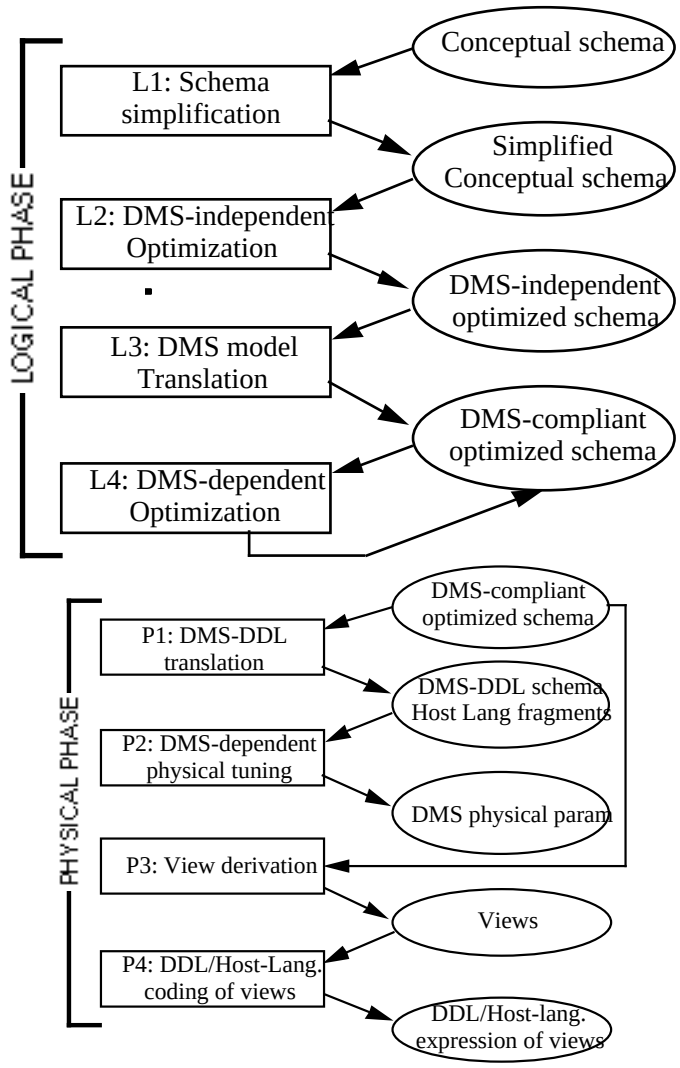


Figure 1.3 *- Some important design processes and products of database design. The processes have been classified into the logical and physical phases.*

Conceptual phase : user requirements are collected, analyzed and formalized into a conceptual schema. Since it has no impact on reverse engineering, this activity is not shown in figure 1.3.

Logical phase : the schema can be expressed (L1) into a simpler model (e.g. Bachman's model), better suited for optimization reasonings; it can be optimized independently of the target DBMS (L2); it is then translated according to the target data model (L3); at this stage, it can be further optimized according to DMS-dependent rules (L4).

Physical phase : the logical schema is expressed according to the DDL[4] of the DMS and to procedural languages (P1); its physical parameters are set through DMS-dependent physical tuning (P2); the view needed by the application programs are derived (P3) and expressed partly into the view DDL, and partly into the host language.

1.4.2 What makes *DBRE* so difficult ?

The final, *executable description* of the database, i.e. the *DDL/Host language* expression of schemas and views, can be seen as the result of a chain of transformations that have ***degraded*** the origin conceptual schema. Let's reexamine each process of the generic model of database design as far as it introduces some degradation into the conceptual schema.

Process L1 : the conceptual specifications are preserved, but they are expressed with poorer structures, leading to a less concise and readable schema. For instance, n-ary relationship types have been transformed into binary ones, multivalued attributes have been reduced to single-valued ones, or IS-A links are transformed into one-to-one relationship types.

Process L2 : the schema can be restructured according to design requirements concerning access time, distribution, data volume, availability, etc. The conceptual specifications are preserved, but the schema is obscured due to non-semantic restructuration, such as structure splitting or merging, denormalization or structural redundancy.

Process L3 : the data structures are transformed in order to make them compliant with the model of the target DMS. Generally, this process deeply changes the appearance of the schema in such a way that the latter is still less readable. For instance, in standard files (resp. relational DBMS) many-to-many relationship types are transformed into record types (tables) while many-to-one relationship types are transformed into reference fields (foreign key). In a CODASYL schema, a secondary identifier is represented by an indexed singular set. In a TOTAL or IMAGE database, a one-to-many relationship type between two major entity types is translated into a detail record type. Frequently, names have to be converted due to the syntactical limitations of the DMS or host languages.

On the other hand, due to the limited semantic expressiveness of older (and even current) DMS, this translation is seldom complete. It produces two subsets of specifications : the first one being strictly DMS-compliant while the second one includes all the specifications that cannot be taken in charge by the DMS. For

---

4 DMS stands for Data Management System (including Database Management Systems or DBMS. The Data Description Language (DDL) is the language of the DMS through which the data structures are declared or defined.

instance, referential constraints cannot be processed by standard file managers. In principle, the union of these subsets includes the conceptual specifications.

Process L4 : the schema is restructured to match design criteria such as access time. This restructuring depends on the specific behaviour of the DMS. Just like process L2, it makes the schema less readable. The semantic contents of the DMS-compliant structure are preserved.

Process P1 : only the first subset of specifications collecting strictly DMS-compliant structures can be translated into the DDL of the DMS. The discarded specifications should be either ignored, or translated into languages, systems and procedures that are out of control of the DMS (e.g. host language, user interface manager, human procedures, etc). From now on, the schema of the database generally represents a strict subset only of the conceptual specifications.

Another phenomenon must be pointed out, namely **structure hiding** (figure 1.4). When translating a data structure into DMS-DDL, the designer (or programmer) may choose to hide some information, leaving to the host language the duty to recover this information. The most widespread example consists in declaring some subset of fields (or even all the fields) of a record type (or segment, or table) as a single, unstructured, field; recovering the hidden structure can be made by storing the unstructured field values into a host program variable that has been given the adequate structure.

Finally, let's observe that the DMS-DDL schema is not always materialized. Many standard file managers, for instance, doesn't offer any central way to record the structure of files, e.g. in a data dictionary.

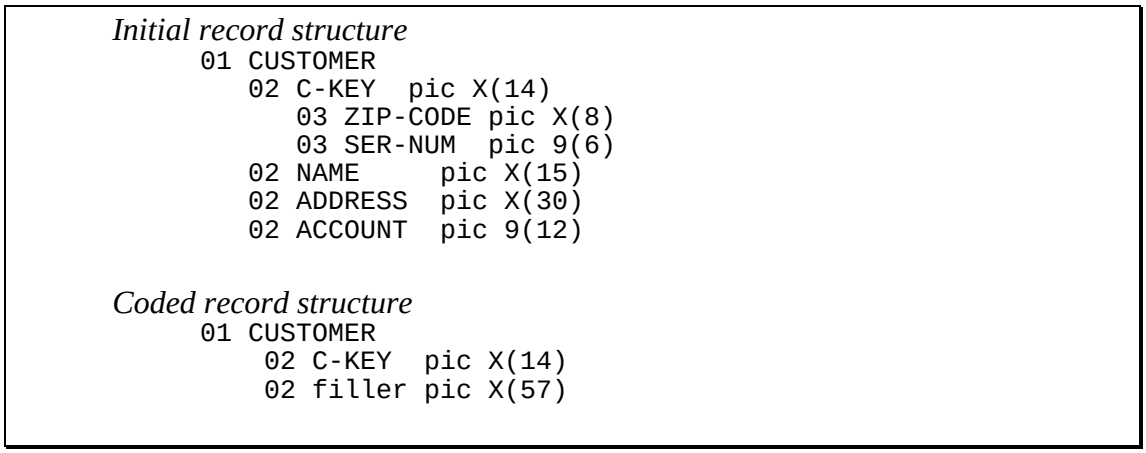


Figure 1.4 *- An example of* ***structure hiding****. The decomposition of both the key part and the data part are replaced by anonymous data structures in the actual coded structure. Though it can simplify data description and data usage, this frequent technique makes data structure considerably more complex to recover.*

Process P2 : works at the internal level, and doesn't modify the schema as it is seen by the programmer.

Process P3 : each view is dedicated to a limited set of application programs (or users). In principle, the set of the views cover all the data structures they derive from.

Process P4 : this process is similar to P1. Each view is expressed into a mixture of DDL texts, host language procedures, screen/report specifications, etc. Here too, the principle of structure hiding can be applied heavily, specially in case of standard file managers. Figure 1.4 is an illustration of this practice. Figure 1.5 shows a common way to translate lost referential integrity constraints into the procedural part of the program.

Observation : in poor DMS, like most standard file managers, **the result of process P4 is the only information that is still available** about the implemented data structures. This fact ***plus*** structure hiding make the reverse engineering of standard file an exercise particularly challenging.

Let's draw some conclusions from this analysis :

- producing an executable schema of a database can be seen as the step-by-step transformation of its conceptual schema.
- each transformation introduces some sort of degradation in the schema that makes it less complete, simple, intuitive and readable.
- the kind of degradation depends on the objective of the design process.

```
fd F-CUST;
 record is CUSTOMER.
 01 CUSTOMER.
    02 CNUM pic X(14).
    02 CDATA pic X(57).

 fd F-ORD;
 record is ORDER.
 01 ORDER.
    02 ONUM pic 9(8).
    02 O-DATE pic X(12).
    02 CUST pic X(14).

working storage section
 01 CN pic X(14).
 01 C.
    02 CNUM pic X(14).
    02 filler pic X(57).
 01 O.
    02 ONUM pic 9(8).
    02 O-DATE pic X(12).
    02 CUST pic X(14).

procedure division (in pseudo-code)
    ...
    read CUSTOMER(CNUM = X) into C
    if found(C) then
          O.ONUM := ...
          O.DATE := ...
          O.CUST := C.CNUM
```

```
            write O into ORDER
      endif
      ...
```

Figure 1.5 *- Expressing non-DMS structural parts (here a referential constraint) by procedural statements. The latter, sometimes centralized into a single file management module, check the non-violation of constraints before data insert, update and delete. These sections are often written according to regular patterns such as that presented here.*

1.4.3 Gross architecture of a *DBRE* methodology

Let's first observe that reverse engineering a database is concerned with the design decisions that have been taken during the logical and physical phases only. The proposed approach is based on playing backward these two phases, starting from the results of the latter one. Grossly speaking, the RE analyst is faced with the following problem :

- given the DDL/host language expression of existing data structures (global schema and/or views),
- given known operational requirements (e.g. the DMS, performance requirements, etc),

Ø find a possible conceptual schema that could lead to these data structures.

The process can be split into two main phases, rather independent from each other :

Ø retrieve the existing data structures from their DDL/host language expression, and

Ø retrieve a possible conceptual schema that defines the semantics that underlies these data structures.

The first process is the reverse of the physical phase, and has been named *Data structure extraction*. The second one is the reverse of the logical phase, and has been named *Data structure conceptualization*. In a first approach, we can consider that they are conducted sequentially. According to the analysis developed above, the gross organization of the method can be sketched as in Figure1.6.

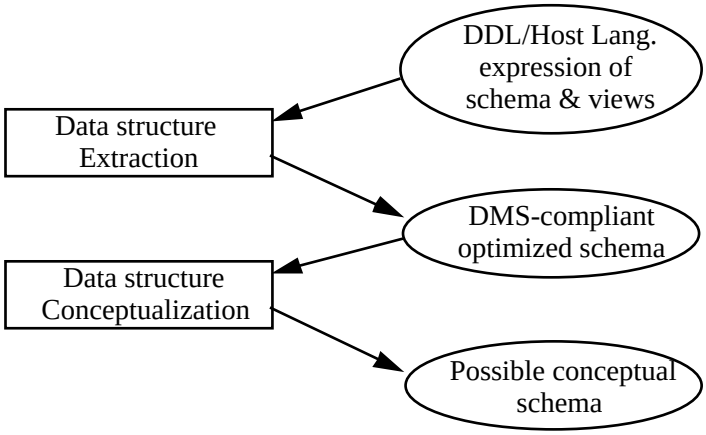


Figure 1.6 *- Gross organization of database reverse engineering activity.*

In low level DMS (e.g. standard file managers), the starting point generally is made up of the DDL/Host language expression of the views. In higher level DMS, the input information generally consists of the DDL expression of the global schema. In both cases, these expressions should be augmented with the translation of discarded specifications. In some organizations, the data structure descriptions are available through a data dictionary system, making the first process a bit easier.

1.4.4 Data structure modeling for *DBRE*

Database design and reverse engineering are concerned with building, converting and transforming database schemas at different levels of abstraction. Elaborating DMS-independent and even methodology-independent techniques and reasonings that support these activities requires the availability of a set of models to express all these schemas. Due to the transformational approach adopted in this presentation, and due to the large scope of the proposal that encompasses all the traditional levels of abstraction, it has been found essential to base it on a unique schema specification model. This model and its transformational operators are intended,

- to support forward as well as reverse engineering,
- to express conceptual, logical and physical schemas, as well as their manipulation,
- to support any DMS model and the production and manipulation of their schemas.

In short, conceptual schemas as well as physical schemas are expressed into a unique, generic, *extended entity-relationship model*. To make the presentation clearer, we shall distinguish two layers of features in the model, comprising the conceptual and technical features respectively. Here follows a brief description of the concepts of the model. A more comprehensive specification can be found in [HAINAUT,92a] and [HAINAUT,92c]; formal foundations of most aspects of the model are proposed in [HAINAUT,89] and [HAINAUT,90].

The **conceptual layer** consists of the features of the standard entity-relationship model with some extensions. It includes the following concepts, most of them being illustrated in Fig. 1.7 :

- *entity type*, comprising any number (including zero) of attributes;

- *IS-A hierarchy* ,
- *relationship type* (called *rel-type* from now on), comprising two or more roles and any number of attributes; a role is taken by one or several entity types (multi-ET role), and is given a cardinality constraint [min-max] that states the minimum and maximum number of relationships in which any entity takes this role;

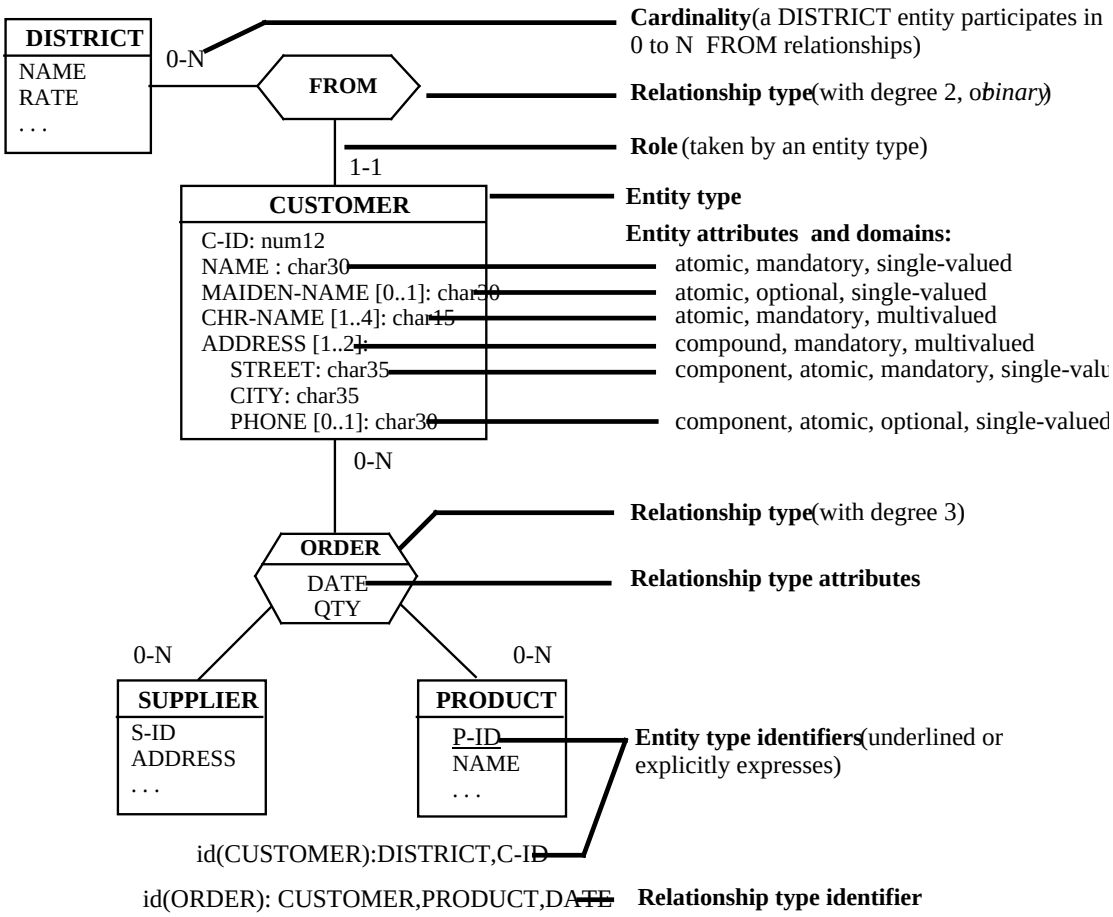


Figure 1.7 *Graphical representation of some conceptual constructs.*

- an *attribute* is either atomic or compound; an atomic attribute has a domain of values; each attribute is given a cardinality constraint[5] [min-max] stating how many values can be associated with its parent object (i.e. entity type, rel-type, compoud attribute); a multivalued attribute (cardinality max > 1) can be pure (set of values), bag (multiset of values) or list (indexed set or multiset);
- an entity type can have any number of (candidate) *identifiers*, including zero; an identifier is made of attributes and/or roles (i.e. connected entity types)[6]; one of the identifiers can be declared primary;

---

5 Note that this constraint allows for the specification of optional/mandatory attributes as well as single-valued/ multivalued attributes. In addition using this constraint for both roles and attributes stresses their duality and simplifies greatly many transformations.

6 In [BATINI,92], identifiers are called *internal* when they comprise attributes only, *external* when they include roles only, and *mixed* when they are made of both.

- a rel-type has at least one identifier made of roles and/or attributes; any role with cardinality [`min-1`] is an identifier; when no identifier is specified or can be deduced, then all the roles of the rel-type form its identifier;
- a pure or list multivalued attribute can be given an identifier which is a subset of its (grand-) children components;
- integrity constraints can be associated with these constructs; let us mention inclusion constraint, referential constraint[7], redundancy constraint, exclusion constraint, coexistence group[8] and functional/multivalued dependency.

The **technical layer** includes constructs that pertain to the description of logical and physical data structures (Figure 1.8). These features are additions to the conceptual layer described hereabove. Some of them are described briefly.

- a file is a repository of entities; at this level, an entity is the abstraction of a record, segment or row;
- an access key is a goup of attributes with which an access mechanism is associated; it is the abstraction of value-based access mechanisms such as indexes, hash files, etc;
- an access path is an access mechanism allowing the navigation through rel-types; it allows to specify set types (CODASYL) and parent-child relationships (IMS) for instance;
- the entities in a file, the entities linked with an entity through a rel-type and list multivalued attribute values can be ordered according to insertion time or to a sort-key;
- an attribute has a physical length, depending on its encoding scheme, and a physical position in its parent object. Two attributes can share the same physical position.

Finally, some procedural concepts that are relevant when dealing with file and record processing are also described in the model. Let's only mention the notions of *application, source file, module, variable, transfer instruction* (MOVE, CALL USING, READ INTO, etc), *condition name*, etc.

---

7 This is a special case of inclusion constraint that is not based on the concept of primary key; it only requires the presence of a (candidate) identifier in the target entity type. This allows a greater simplicity in the reverse engineering untranslation algorithms as compared with proposals such as [NAVATHE,88] and [FONKAM,92].

8 A group of attributes and/or roles of an entity type whose values are simultaneously present or absent. This concept has been formalized in [MATHERON,91].

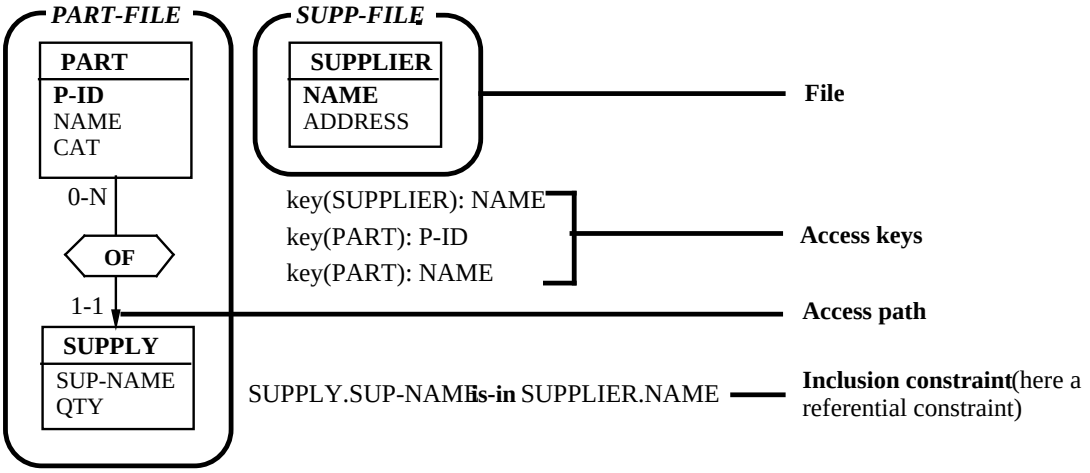


Figure 1.8 *Graphical representation of some technical constructs. The figure illustrates also the notion of inclusion constraint*

This model must be considered as a generic specification model that can be specialized into a great variety of submodels, for instance according to the design levels or design product classes of a standard or user-defined multilevel design method, or according to the target DBMS. Specializing the generic model into a specific submodel can be done by stating the set of rules the schema must satisfy.

For example, several methodologies will consider *ER-compliant*[9] a schema that satisfies the following rules :

- Entity types have at least one attribute;
- Attributes are single-valued, mandatory (cardinality [1-1]) and atomic;
- Each entity type has one and only one identifier;
- An entity type identifier is made of one attribute;
- The cardinality constraints allowed are 0-1, 1-1, 0-N, 1-N;
- A relationship has no explicit identifier;
- There are no technical constructs.

Similarly, in an *ORACLE/V5-compliant* schema ,

- there are no rel-types;
- there is from one to 254 attributes per entity type;
- the attributes are single-valued and atomic;
- an identifier must be an access key (i.e. an index);
- an entity type cannot be in more than one file.

---

9 This corresponds to a rather simple model that can be found in many methodologies and CASE tools.

- a name is made up of 1 to 30 characters, the 1st being a letter, and the other ones being letters, figures, '_', '$' or '#';
- a name cannot belong to the ORACLE reserved-word list ('create', 'index', etc);
- the total length of the attributes that make an access key cannot exceed 240 char.

In these rules, an entity type is to be interpreted as a *table*, an attribute as a *column*, an access key as an *index*, a file as an ORACLE *space*.

1.4.5 Data structure extraction

The data structures of the DMS/Host language optimized schema, are found out by analysing the source code of the schemas and programs. The result of this process is a (hopefully) complete, formalized, model of the physical data structures, as they are perceived by users and programmers, according to the DMS model. For some DMS, the analysis is rather straightforward (e.g. CODASYL, IMS or relational), while for others, it requires considerable work and knowledge (e.g. standard files).

The problem is twofold :

- retrieve the explicit and implicit (hidden) data structures that are under the control of the DMS,
- retrieve the discarded specifications (generally integrity constraints ignored by the DMS).

1.4.5.1 Retrieving the DMS data structures

The difficulty of the first problem is dependent on the DMS. In true DBMS, the description of the global DMS-compliant schema is generally available, either as a DDL text, or in a data dictionary (e.g. *system catalog tables*). In lower-level DMS, such as standard file managers, the problem can be much more complex. Indeed, the global schema, generally file and record structures, is not under the control of the DMS, and is only available in the (generally lost or obsolete) documentation of the application. The only description available is made up of file and record descriptions included in the source programs. Since these descriptions can be, and generally are, partial descriptions only, reverse engineering the data structures used by a source program produces one view of the global schema only.

Two difficulties must be solved when carrying out this process : (1) an application program uses several files, a subset only of which concern the database; print, output, temporary, update or sort files must be recognized and discarded[10]; (2) due to the structure hiding habits of many programmers, the actual structure of a record type or of a field can only be elicited by analyzing how and where records/fields are used; this

10 Unless these files may give hints on the schema when they store data from the database.

means for example examining the control flow of the procedural parts of the program, as illustrated in figure1.9, or analyzing the entry forms and the output reports/forms.

```
 program file
      01 CUSTOMER
          02 C-KEY  pic X(14)
          02 filler pic X(57)

 working storage section
      01 IN-CUST
         02 ZIP-CODE pic X(8)
         02 SER-NUM  pic 9(6)
         02 C-DATA   pic X(57)

      01 EXPLODE1
         02 NAME     pic X(15)
         02 ADDRESS  pic X(30)
         02 ACCOUNT  pic 9(12)

procedure division
      ...
    read CUSTOMER into IN-CUST.
      ...
    move C-DATA of IN-CUST into EXPLODE1.
      ...
```

Figure 1.9 *- Hidden structure elicitation. The original record decomposition is recovered through data flow analysis.*

1.4.5.2 Retrieving discarded specifications

This process is mainly based on the analysis of the procedural parts of the applications, be they program sections or triggered actions associated with user interface forms or with database events. These procedures mainly check integrity constraints and compute derived data. Their structure is generally simple and standard, and can be recognized easily, provided we can locate them in the huge amount of source code. Among the main constraints the texts have to be searched for, let's mention the referential constraints (in standard files and in relational databases), the identifiers (in sequential files and tables, in network databases), one-to-one relationship types (when many-to-one only are available), exclusive structures and redundancy constraints.

In some circumstances, the data themselves have to be analyzed. Indeed, evidence of uniqueness properties (identifiers), referential integrity, fine-grained field decomposition or value domain, can be found out by careful examination of the data.

1.4.5.3 Integrating views

When the previous processes have recovered views of the data structures only, e.g. in low-level DMS, two strategies are possible :

- integrate these views to obtain a global schema, then conceptualize it;
- conceptualize each view, then integrate the conceptual views obtained in this way.

The first strategy allows to reduce as early as possible the number of schemas to be conceptualized (collecting more than 100 views is not uncommon in large applications). In addition, powerful heuristics can be used at this level, based on position and length of data fields in the records, multi-record type files or the use of compile-time statements (e.g. COBOL *copy* or C *include*). However, integrating physical views can prove complex when they include many performance-oriented constructs such as redundancies, denormalized structures and other tricks. The second strategy implies conceptualizing a larger number of (often similar, if not identical) views, and dropping useful hints from the physical structures. However, view integration is easier, since it corresponds to processes that are now standard[11] [BATINI,86], [BOUZHEGOUB,90], [SPACCA,92].

1.4.6 Data structure conceptualisation

The objective of this process is to discard technical constructs from the DMS-compliant schema, to reduce DBMS-dependent constructs, to eliminate performance-oriented data redundancies, to make hidden conceptual data structures explicit, and to produce a clear, normalized and natural conceptual schema.

1.4.6.1 Schema cleaning and renaming

The schema may include constructs that do not pertain to the database itself. Structures for transient data, state variables of programs, as well as dead parts must be detected and eliminated. This process requires not only domain knowledge, but also information on the structures of the programs. In addition, the name of the data structures may need translation for various reasons : weakness of the DMS syntax conventions, reserved words (DMS, host language, local standard), multi-lingual naming, heterogeneous and undisciplined naming conventions, etc.

---

[11] There are currently no literature on integrating physical data structures.

1.4.6.2 Elimination of DMS-specific optimization constructs

The schema is examined for optimization constructs that are specific to the origin DMS[12]. A deep knowledge in the physical behaviour of the DMS is required. Some examples : field padding for address alignment, record splitting when the size is greater than page size, grouping frequently accessed records in the same database space stored in a high-speed device, defining non-information-bearing set types in CODASYL databases, etc. These constructs must be recognized and discarded.

1.4.6.3 *Un-translating* the DMS-compliant schema

Producing a DMS-compliant schema implies translating the data structures that are not supported by the DMS. Detecting such transformed structures, and replacing them by their conceptual origins lead to a higher-level schema. A good knowledge of the forward transformation rules generally used when translating into the DMS model is necessary. The biggest problem is that there is generally no 1-1 mapping between the conceptual structures and their DMS-compliant translation. A conceptual construct can be translated into several DMS structures, while several different conceptual constructs can be translated into the same DMS structure.

1.4.6.4 Elimination of DMS-independent optimization constructs

Some optimization practices are valid whatever the target DMS. Examples : merging/splitting record types, derived attributes, denormalization. Recognizing them allows one to discard them.

1.4.6.5 Expressing the schema in a higher-level model

Eliminating optimization-oriented structures and retrieving the conceptual origin of each construct of the DMS schema can produce a schema that is still awkward and unclear. By restructuring this schema, it is possible to make it more readable and more natural. Replacing binary structures by n-ary ones, extracting complicated compound multivalued attributes to replace them by entity types, defining generic entity type by factoring similar semantic properties of a set of entity types, generating specific entity types from optional, exclusive attributes and roles, are some examples of transformations that can help to meet these objectives.

---

[12] In all generality, performance and technical efficiency are only examples of requirements that can lead to schema restructuring. Availability, security, data modularity, distribution, etc, are other examples of requirements that will shape the final schema. Being aware of these requirements should ease the reverse engineering process.

1.4.6.6 Integrating schemas

When the views extracted from the source programs have been conceptualized, they should be merged once they have reached this state. Schema integration will also occur when reverse engineering an information system consisting of more than one database, or a heterogeneous database (such as an IMS database + VSAM files).

1.4.7 *DBRE* techniques

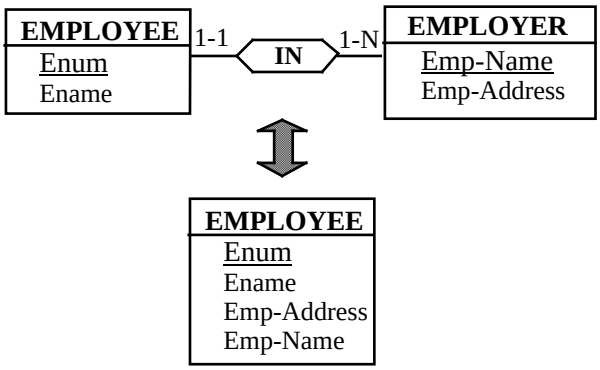
Several major DBRE processes are based on a limited set of common techniques. We shall describe shortly three of them, namely *schema transformation*, *redundancy elimination* and *schema integration*.

1.4.7.1 Schema transformation

Schema transformation is the basic tool in many database design activities. An in-depth discussion of the concept can be found in [HAINAUT,91] and [HAINAUT,92a]. Grossly speaking, a transformation consists in replacing a data structure with another one which has some sort of equivalence with the former. The most important equivalence that is sought is semantic equivalence. In this case, both schemas express exactly the same semantics; in addition such a transformation is said to be reversible, i.e. it exists another, inverse, transformation that transforms the final schema into the former one.

In most cases, the final structure satisfies a criterion the former doesn't meet. DMS compliance, space or time efficiency, normalization are such criteria.

In the database forward engineering model we rely on, the final schema is obtained by successive transformations of the conceptual schema. When transformations guaranteeing semantic equivalence have been used, reverse engineering their resulting schema can be done by using the inverse transformation.



Emp-Name --> Emp-Address

Figure 1.10 *- Read top-down, the transformation denormalizes the schema, leading to better access performances, or decreasing the number of entity types for instance. It is a typical design transformation that can be used in activities L2 and L4. Read bottom-up this transformation eradicates a*

transitive FD, and therefore normalizes the attribute structure of EMPLOYEE. It is a typical reverse engineering transformation.

We shall illustrate the concept with an example of transformation that eliminates a transitive functional dependency that holds into the attributes of an entity type (Figure 1.10), and with another example that tends to make a schema *more relational* (Figure 1.11). Both transformations ensure semantic equivalence.

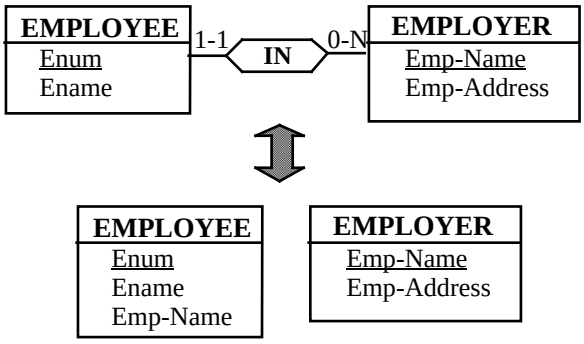


Figure 1.11 *- Read top-down, this transformation allows the representation of many-to-one relationship types with reference attributes, a construct that is compatible with relational databases and standard files (used in activity L3). Read bottom-up, it allows the explicitation of relationship types in a reverse engineering activity.*

It can be shown that a fairly small number of standard transformations can explain how most optimized DMS-compliant schemas encountered in practice have been obtained. Understanding their mechanism, and how they relate with optimization and translation reasonings is essential for reverse engineering complex schemas.

1.4.7.2 Data redundancy elimination

Introducing data redundancies in a database schema is very common. The main objectives are better access performances, higher availability or recovery (they are used in processes L2 and L4). Recognizing redundant structures, and understanding their fundamental mechanisms is important in the reverse engineering activities.

There are two basic techniques for defining redundancies in a schema :

- through **structural redundancy**, a new object type B is added into the schema in such a way that instances of B can be computed from instances of other object types of the schema. Examples : attribute Total-Amount in entity type ORDER, attribute Number-of-Employees in entity type EMPLOYER.
- **denormalization** consists in grouping independent fact types in such a way that their instances are available simultaneously. This construct reduces the number of

aggregates (entity or relationship type) and may decrease the access time. See example in Figure 1.10.

1.4.7.3 Schema integration

Schema (or view) integration is a design domain that studies the merging of specifications, i.e. schemas, whose real worlds may overlap. This merging is not a pure addition of these source schemas since each fact/object of the real world must be represented only once in the resulting schema. Given a collection of source schemas, several integration strategies have been proposed : merging two schemas at a time or all the schemas in parallel, producing a new schema or augmenting one of the source schema. Binary integration is generally carried out in four steps :

- preparation of the schema (optional) : restructuration of some schemas in order to make the merging easier; in some techniques, it allows to automate the next two steps;
- correspondence : an object type in one schema is related to an object type in the other schema to state the similarity of the real world objects/facts they describe. The kind of this relationship is stated : they are the same, one is included in the other one, they have a common generic type, etc;
- merging : the object types in correspondence are merged, together with their relationships with the other object types of the schemas;
- restructuration (optional) : the final schema is refined in order to make it simpler and more readable.

In reverse engineering, schema integration can occur at different levels. At the conceptual level (merging conceptualized views), the problem is fairly standard. At lower levels, such as integrating DMS-compliant views, the problem can be somewhat more complex, because the schemas include non-semantic constructs.

1.4.8 Specific *DBRE* methodologies

The principles that have been presented so far are not dedicated to specific RE approaches, nor to specific DMS. With this respect, they must be perceived as a generic framework in which such specialized approaches can be defined. It is fairly easy to state the dependency of each RE process on specific contextual aspects. For instance,

- processes 1.4.5.1, 1.4.5.2, 1.4.6.1 depend on corporate programming standards,
- processes 1.4.5.3, 1.4.6.6 depend on the number and similarity of the views,
- processes 1.4.5.1, 1.4.5.2, 1.4.5.3, 1.4.6.1, 1.4.6.2, 1.4.6.3 depend on the DMS,
- processes 1.4.6.2, 1.4.6.4 depend on optimization requirements,
- process 1.4.6.5 depends on corporate analysis standards.

Specializing these processes according to these criteria provides us with a specific DBRE method. It should be noted that this specialization can rely mainly on the genericity of such components as the unique data model and of the schema transformations. For instance, specialization according to the DMS can be defined by specializing the data model (it can be limited to the structures known by the DMS), and by selecting the schema transformations that could have been used by the database designer for translating into this DMS. Such methodological specialization has been studied in [HAINAUT,92a] for DB forward engineering.

1.4.9 Conclusions

Except in oversimplistic, or accidently favourable situations, reverse engineering a database from source DDL/Host language texts is a complex task that still needs in-depth research. Understanding formal and empirical design methodologies, together with their underlying techniques and reasonings, is a considerable asset in this process, mainly because it helps to identify the main design processes that shape the final schema by including specific requirements. A careful analysis of that *shape* can give hints as to the conceptual/technical/organizational origin of the observed data structures. The paper establishes a general framework for understanding practical (i.e. mainly intuitive and non formal) design methodologies and to analyse the possible ways a conceptual schema could have been transformed into the observed schema.

It must be clear that reverse engineering cannot be fully automated process. Indeed, it must integrate not only formal knowledge on database modeling and design, technical knowledge on the implementation tools, but also knowledge on how programmers program(med), how designers design(ed), as individuals (through their personal behaviour) and as members of an organization (following possible methodological standards). In addition, knowledge on the application domain, together with information from other sources (obsolete and incomplete documentation, data dictionary, file contents, etc) are generally used to support the source text analysis.

1.5 ORGANIZATION OF THE MANUAL

This manual develops a rigourous framework of concepts and processes aimed at reverse engineering the data structures of data-oriented applications. It is organized in five main parts.

Part I : Introduction to database engineering

- Chapter 2 introduces the reader to the principles of database modeling. It presents a variant of the Entity-Relationship model, the data structure model that will be used throughout the reverse engineering activities.
- Chapter 3 proposes an in-depth analysis of the database design process, and of all its design activities. It discusses the objectives and reasonings that allow the production

of an operational description of a database from user's requirements. In addition, a description model is suggested to describe design methods.

Part II : A general model of database reverse engineering

- Chapter 4 introduces the problem of database reverse engineering by describing the various objectives of this process.
- Chapter 5 defines a general model of the database reverse engineering activity. This model derives from the idea that reverse engineering can be perceived, at least at a theoretical level, as the reverse of database design. Some specific problems are introduced. They will be developped in the following chapters.

Part III : Specific problems of database reverse engineering

- Chapter 6 discusses the different aspects of name processing in database reverse engineering.
- Chapter 7 is a central section of this manual. It develops an important formal tool for database engineering, and particularly for reverse engineering : schema transformations.
- Chapter 8 is a synthesis of the problem of representing real-world facts by data and data types.
- Chapter 9 develops a particular aspect of the representation problem : data redundancy. It analyses its different forms, and discusses the ways to get rid of redundancies. An important section is dedicated to the normalization problem, particularly in *non-flat* data structures.
- Chapter 10 describes the problems and solving techniques related to the multiplicity of descriptions. How to get a complete, unique, schema of a database when we are provided with several (partial) descriptions of it ?

Part IV : Database reverse engineering methodology

- Chapter 11 (and 12) go back to the fundamental reverse enginering methodology proposed in chapter 5 by developing the first process : data structure extraction, through which the technical data structures of the source database are made explicit.
- Chapter 12 develops the second major process : data structure conceptualization, through which the semantics of technical data structures are elicited.
- Chapter 13 describes the links that should exist between the different levels of description of the data.
- Chapter 14 proposes a short discussion on some important strategic aspects of database reverse engineering.

Part V : Miscellaneous

- Chapters 15 to 17 illustrate the concepts presented in this manual through the resolution of small case studies of SQL, COBOL and CODASYL databases.
- Chapter 18 collects the bibliographical references.

In addition, a companion volume will include the technical material that complements the general propositions of this manual.

Chapter 3

DATABASE FORWARD ENGINEERING

This chapter presents a generic database design methodology based on a multi-step schema transformation and enrichment. Each step is dedicated to a design criterion and is made up of specific activities. This analysis will allow to locate and specify the main decisions that contribute to the production of an operational database schema. This analysis will be the basis for developing a systematic reverse engineering methodology.

3.1 INTRODUCTION

Database reverse engineering is an activity that needs a deep understanding of the forward process, it's to say database design.

In all generality, database design is a semi-formal[1] process the ultimate aim of which is to create an efficient, operational database that is a reliable model of an application domain.

Let's detail this aim a bit further.

- *The database is a reliable model of the application domain*

  The database must represent all the facts of the application domain that are of interest for the intended users. It must represent these facts in a natural way, but also in a stable way, that is such that slight variations in the application domain will not make

---

[1] and therefore semi-creative, a fact that will account for the major difficulties we will meet in reverse engineering.

the structure of the database obsolete. As far as possible, it will satisfy the basic principle that states that each fact is recorded only once.

- *The database is operational*

  The database must be managed by computer-based resources such as a DMS. Consequently, its data structures must comply with the characteristics of these resources. For instance, using a relational DBMS implies that the database schema is in a flat table format.

- *The database is efficient*

  The database must deliver data and manage them with good performances. In addition, ensuring qualities such as availability, smooth data sharing and privacy is another kind of required performance.

3.2 MULTI-APPROACH TO DATABASE DESIGN

To achieve these contradictory goals, database design has long been tackled through multi-level approaches, where each level is dedicated to a family of objectives. Indeed, decoupling these objectives leads to more effective and reliable design processes.

Adopting the old programming proverb stating that *it is easier to optimize a correct program than correcting an optimized one*, all design methodologies propose building a correct specification of the database, independently of the computer resources, then implementing it in an operational DMS in an optimized way.

A general framework for database design[2] could be organized as follows ([BODPIG,89] and [HAINAUT,86]).

1. *Building the conceptual schema of the database*

   Through this process, the application domain is analysed and the conceptual schema of the future database that best models it is built. This schema is normalized in order to give it a clean and non-redundant structure. The only quality of the conceptual schema is that it is a faithful and natural model of the application domain[3].

2. *Building a logical schema of the database*

   The conceptual schema is translated according to the data model of the target DMS. The fine tuning is not defined yet, but some structural transformations can be carried out in order to optimize data processing. This results in a logical schema, which is a (generally optimized) DMS-compliant schema.

---

[2] due to the objective of the manual, this framework has been somewhat simplified. In particular, the conceptual design process has been given a lesser importance than it has in reality.

[3] let's remind that this sentence means : *this schema defines data structures that make a faithful and ...*

3. *Defining the physical schema of the database*

   The logical schema is translated into the DDL of the DMS. Physical parameters are evaluated in order to get an efficient database.

Note

The steps mentioned above concern the design of the whole database. A last step generally occurs, defining the users view, i.e. the subschema the user will be provided with for accessing the database.

3.3 CONCEPTUAL DESIGN

This process consists in several subprocesses that can be sketched as follows.

3.3.1 Decomposition of the application domain

In complex situations, the application domain is naturally structured into subsystems. Each of them is characterized by a high level of internal consistency (human relationships, objectives, resources, objects and activities, vocabulary, standards, etc). It is expected that analysing a subsystem is much easier than analysing the whole application domain.

3.3.2 Building the preliminary conceptual schema for each subsystem

The subsystem is analysed in details in order to elicit the fundamental facts that make the basis of its activities[4]. These facts are structured in terms of objects, object properties and relations between objects.

Translated according to the Entity-Relationship model (or in any equivalent conceptual model), these facts are expressed through the so-called conceptual schema of the subsystem[5].

There are generally two strategies for building a conceptual schema.

- The **first strategy** consists in trying to discover the main entities of the subsystem, then finding the relationships between them, and finally refining their description by finding their properties.

---

[4] we limit intentionally our scope to the facts that will lead to the definition of data structures of the future information system. Wider design procedures can be found in the literature [BOD-PIG,89].

[5] in fact, it is one of the possible conceptual schemas. Ideally, all the possible conceptual schemas are equivalent. Therefore, choosing the best one is a false problem.

- The **second strategy** consists in building the schema incrementally, according to the following procedure : at a given point of time, we have a partial conceptual schema and a set of facts that have not been analyzed yet. One fact is chosen, and is compared with the current state of the schema. If the fact is already represented, it is discarded. If the fact is new, an adequate structure is added to the schema. This structure can be an new entity type, a new relationship type between existing entity types, an new attribute or a constraint. In some cases, the fact can be represented after a reorganization of the current schema (e.g. an attribute or a relationship type is transformed into an entity type). The fact can also be in contradiction with the schema. In that case, the conflict must be solved. This conflict-solving problem is a special case of schema integration[6], a domain that is related to several activities in database design (both forward and reverse engineering). This problem will be developed in chapter 10.

According to some authors [BOD-PIG,89], better results can be expected from the second approach.

The main sources of information from which the relevant facts can be found are user interviews, analysis of the organization, of its information flows, of its activities, the current documents, the existing information system[7].

3.3.3 Normalization of the preliminary conceptual schemas

Each schema is examined in order to detect and correct representation problems such as,

- redundant structures : the same fact is represented more than once, or a fact represented can be derived from another one;
- unnormalized entity types and relationship types : a functional dependency where the left-side member is not an identifier; presence of non-trivial multivalued dependencies;
- lack of generality : the orders of a customer are represented by a multivalued attribute;
- readableness problems : a relationship type is represented by attributes referring to an entity type (*à la* relational);
- unnecessarily complex structures : entity types bearing no semantics; simple attributes represented by entity types;
- violation of naming conventions : two relationship types have the same name.

---

6 the new fact can easily be represented by a small schema, which is to be integrated into the current schema.

7 this source is widely used in practice, at least for input-output documents and file structures. This means that reverse engineering is an integral part of some approaches of forward engineering.

Most of these problems can be dealt with by using schema transformations. These operators are relevant in most database design activities, be they related to forward or to reverse engineering. Due to the very objective of this manual, the description of schema transformations will be developed in chapter 7.

Some of the schema transformations that can be of interest in this step are the following.

- transformation of a set of attributes of an entity type into a new entity type + a relationship type;
- transformation of set of attributes (+ referential constraint) into a relationship type;
- transformation of an entity type into attributes;

3.3.4 Integration of the preliminary conceptual schemas

The normalized schemas of the subsystems are integrated in order to produce a unique, global conceptual schema.

The process of schema integration is complex and will be a major tool in reverse engineering. It will be studied in chapter 10.

3.3.5 Validation of the global conceptual schema

The resulting schema must be validated by users. They have to check whether these specifications represent adequately and completely their needs. This validation can be coupled with that of the specifications of data processing. For instance, one can check that all the data needed by the programs can be found somewhere, specially in the database. The most popular way of validating a database is through prototyping. Most current CASE tools can generate SQL schemas from E-R schemas. Though these schemas are

3.4 LOGICAL DESIGN

3.4.1 Principles

The conceptual schema will be processed in order to give it two qualities it is lacking so far : DMS compliance and optimization. Satisfying these two objectives will lead to more or less severe degradations of the clarity and readableness of the schema. The logical schema that results from the logical design must keep satisfying the initial goal of the conceptual schema : to be a faithful model of the application domain. Therefore, the logical design step is supposed to preserve the conceptual structure. Consequently, this

step is generally perceived as carried out through a sequence of **semantics-preserving transformations** of the conceptual schema. In this transformational approach of logical schema production, it must be understood that the expression formalism is the same as in the conceptual level. We will therefore adopt the extended Entity-Relationship model described in 2.6.

In computer-aided database design, the set of needed transformations is fairly small. Indeed, some twenty elementary transformations are quite sufficient to translate any Entity-Relationship schema into a schema that is both optimized and DMS-compliant, whatever this DMS [TRAMIS,90]. In less formal approaches, such as those that are of interest in reverse engineering old files and databases, a much greater variety of transformations can be used. This set is generally limited only by the imagination of the analyst or of the programmer.

One fact must be accepted : there is no one-to-one relation between conceptual schemas and *X*-compliant logical schemas (where *X* is any DMS). This fact derives from two observations :

- there are generally several (if not many) ways to translate a given conceptual construct into *X*-compliant constructs; for instance, a multivalued attribute can be replaced, in a relational schema, by a table, by a list of similar attributes, or by a single, concatenated attribute.
- several conceptual constructs can be translated into the same *X*-compliant construct; for instance, a table can be the relational representation of an entity type, a many-to-many relationship type or a multivalued attribute.

Though they cannot always be distinguished in practice, let's consider each objective of the logical step separately, namely DMS-compliance and optimization.

3.4.2 Translation of the conceptual schema into a DMS-compliant schema

The basic problem is to convert all the constructs of the conceptual schema that have no direct correspondence into the data model of the DMS. Each of the constructs of the conceptual schema is analysed according to the structures allowed by the DMS.

The process can be formalized as follows :

1. search the schema for non-compliant constructs. This searching makes use of the definition of the model of the DMS as explained in 2.4.4 : any construct that violates a definition rule of the model is non-compliant.
2. for each non-compliant construct, choose an equivalent construct to replace it. The replacement construct must be DMS-compliant, or at least, it can be further replaced with a compliant construct (one-step or multi-step transformation).

The problem of schema transformation raises two important questions, namely,

1. what is an equivalent construct ?
2. what replacement construct must be chosen ?

The first question is quite general, and will be examined in chapter II.7. In short, construct S2 is equivalent to S1 if :

1. any data instance s1 according to S1 can be given a data instance s2 according to S2, such that there exists a function or a procedure that can transform s2 into s1 (semantic equivalence);
2. any access mechanism defined in S1 has been transformed into access mechanisms that allow for similar performances. For instance, an access path between two entity types can be replaced with an access key associated with one of them;
3. any additional information associated with S1 is given an equivalent version into S2. Some examples are : textual description, statistics, update rate, etc.

Answering the second question supposes that we have criteria against which we can evaluate each solution. Criteria at the logical level are in terms of performances, availability, distribution of data, modularity, reliability and access control.

Let's consider an example. When producing a relational-compliant schema, a multivalued attribute, such as PHONE-NUMBER[0-5] (an attribute of, say, CUSTOMER), must be replaced by a relational-like construct[8].

```
entity-type CUSTOMER ( CUS-NUMBER: char(8),
                       PHONE-NUMBER[0-5]: char(15),
                       ... )
```

- A first solution is to define a new entity type PHONE with two attributes : PHONE-NUMBER et CUS-NUMBER, the latter being a reference to a CUSTOMER entity.

```
entity-type CUSTOMER ( CUS-NUMBER: char(8),
                       ... )

entity-type PHONE (    PHONE-NUMBER: char(15),
                       CUS-NUMBER: char(8))

PHONE.CUS-NUMBER is-in CUSTOMER.CUS-NUMBER
```

- Another solution is to define five single-valued attributes PHONE-1, PHONE-2, .., PHONE-5.

---

[8] an entity type with atomic, single-valued attributes can be considered relational-like or relational-compliant since it can be given an immediate expression as a table and its columns.11

```
entity-type CUSTOMER ( CUS-NUMBER: char(8),
                       PHONE-1: char(15),
                       PHONE-2: char(15),
                       PHONE-3: char(15),
                       PHONE-4: char(15),
                       PHONE-5: char(15),
                       ... )
```

- A last solution could be a single-valued attribute PHONES, each value of which is made up of the concatenation of the PHONE-NUMBER values of each CUSTOMER.

```
entity-type CUSTOMER ( CUS-NUMBER: char(8),
                       PHONES: char(75),
                       ... )
```

If no access is needed through this attribute, the last transformation will generally be preferred. On the contrary, if PHONE-NUMBER is required to be an access key, the first transformation will be better.

An important aspect of DMS-compliant schema production is adapting the conceptual and logical names to the **naming constraints** of target DMS. Rules concerning the syntax, the character sets or reserved words will often imply changing the name of some objects in a schema.

3.4.3 Optimization of a logical schema

A DMS-compliant schema can be further transformed in order to gain additional properties such as increased performances or availability. We can distinguish three main techniques to reach such objectives, namely schema restructuring, structural redundancy and denormalization.

Schema restructuring consists in transforming constructs without introducing redundancies. Some examples :

1. splitting an entity type into two parts, one collecting frequently updated attributes and the other gathering more stable ones;
2. merging two entity types associated with a one-to-one relationship type in order to get both with a single access;
3. transforming a long attribute into an entity type to decrease the size of the origin entity type.

Structural redundancy consists in duplicating some data types, or introducing some derived data types. This redundancy is characterized by the existence of two different constructs in the schema.

Some examples :

- though the customer address can be obtained through the PASSES relationship type, a (redundant) attribute CUST-ADDRESS is added to the ORDER entity type;
- though it is possible to know all the products purchased by a customer by scanning his ORDERS, then their LINES, and finally reading the corresponding PRODUCTS, a direct (redundant) relationship type is defined between CUSTOMER and PRODUCT.

These redundancies must be associated with redundancy constraints (see 2.3.11.6). They will imply increased update costs and disk space but increased access performances as well.

A **denormalization transformation** consists in merging connected data structures. The redundancies are not visible directly in the schema, but occur at the instance level. This process is the converse of the normalization transformation (see 9.9).

Let's consider an example stating that *each product belongs to a category, which in turn implies a VAT rate.*

A normalized schema will include entity types PRODUCT and CATEGORY (with identifying attribute CODE and attribute VAT-RATE) and the BELONGS-TO(PRODUCT, CATEGORY) relationship type.

```
entity-type CATEGORY ( CODE: ..,
                       VAT-RATE: ..)

entity-type PRODUCT ( PNUM: ..,
                      NAME: ..)

rel-type BELONGS-TO([0-N]: CATEGORY,
                    [1-1]: PRODUCT)
```

In order to decrease the number of entity types and to avoid access to CATEGORY from PRODUCT, it can be decided to merge both entity types by adding CODE and VAT-RATE to PRODUCT, and by getting rid of CATEGORY.

```
entity-type PRODUCT( PNUM: ..,
                     NAME: ..,
                     CODE: ..,
                     VAT-RATE: ..)
```

The resulting schema is unnormalized since there exists a functional dependency from CODE to VAT-RATE, the left-side part of which is not an identifier of PRODUCT. Therefore, the information stating that this code implies this VAT rate will be duplicated as many times as there are products in this category.

Here again, these redundancies must be accompanied by redundancy constraints (see 2.3.11.6). They will imply increased update costs, update anomalies and increased disk space but will increase access performances as well.

3.4.4 Quantifications and access mechanisms

Obviously enough, trying to get high performances, low disk space, high availability or maximum access throughput, cannot be done by examining the conceptual schema alone. Additional information must be collected on the data and on their usage. Three categories of information are needed at the logical design level, namely static statistics, dynamic statistics and usage needs, the former two defining the data and usage quantification.

The **static statistics** concerns the average size of the entity, relationship and attribute populations : how many CUSTOMER entities ? How many ORDER entities for each CUSTOMER entity through PASSES ? How many CUSTOMER entities having a given value of CITY-NAME ? These informations could have been collected at the conceptual level, since they directly relate to application domain facts.

Usage needs define what kind of operations will be performed by the application programs and the transactions. These operations concern mainly updating the data (create, delete CUSTOMER entities, create, delete BUYS relationships, modify attribute PRICE of PRODUCT entity type) and accessing the data (get all CUSTOMER entities, get the CUSTOMER entity linked with a given ORDER entity, get all the CUSTOMER entities with a given value of CITY-NAME).

Dynamic statistics measure the usage needs. For each operation these statistics give the average number of activations per time unit (how many new CUSTOMER entities per day ? How many times CUSTOMER entities are asked for through CITY-NAME ?).

3.4.5 Logical design strategies

Conducting a logical design process is a complex task. In some cases, computer-based tools (CASE tools) can help in this level. Some tools propose a straightforward translation in which the designer has no control. The DMS schema is automatically produced from the conceptual schema. Many of such tools produce DMS-compliant and correct (i.e. preserving the initial conceptual constructs) schemas. However, they cannot produce but simplistic schemas where few or none optimization aspects are tackled. Such schemas are easy to reverse engineer, but generally lead to poor performances in real-world applications. Other CASE tools try to help designers in building correct, DMS-compliant and efficient schemas. They offer evaluation tools, sophisticated transformation tools and expert-system components.

We shall consider here two strategies that can be conducted manually, as it is frequently the case in older applications to re-engineer. These strategies are somewhat theoretic, but allow us to locate where important decisions can have been taken.

Logical design - first strategy : early optimization / late translation

This strategy is based on the early optimization of the conceptual schema. An optimized DMS-independent schema is produced. It is then translated into a DMS-compliant schema whose optimization is further refined if necessary.

1. Transform the conceptual schema into a binary schema by reducing relationship types with a degree greater than two, or with attributes. Motivation : a binary schema is easier to further process and to enrich with quantification and access needs.
2. Define the static statistics.
3. Analyse the data usage needs. The most important applications are analysed. For each of them, the operations needed are found.
4. Define the dynamic statistics. For each operation and for each application that uses it, the activity rate is evaluated. Consolidating these figures gives for each construct the activation rate according to each operation.
5. Optimize the binary schema according selected criteria. Restructuring, redundancy adding and denormalization will lead to new schemas that must be evaluated according to the criteria. The files are defined. The best schema is retained. Note that at this level, this optimization is still DMS-independent.
6. Translate the optimized schema into a DMS-compliant schema.
7. DMS-dependent optimization. Check whether the resulting schema is still optimized. Add some post-optimization to take into account specific characteristics of the DMS and of the computer environment in general. The schema is then ready for a DDL generation.

Logical design - second strategy : early translation / late optimization

This strategy is based on the early translation of the conceptual schema into a DMS-compliant schema. Optimization is carried out on the DMS schema.

1. Transform the conceptual schema into a binary schema by reducing relationship types with a degree greater than two, or with attributes.
2. Define the static statistics.
3. Translate the schema into a simple DMS-compliant schema.
4. Analyse the data usage needs. The most important applications are analysed. For each of them, the DMS-specific operations needed are found.

5. Define the dynamic statistics. For each operation and for each application that uses it, the activity rate is evaluated. Consolidating these figures gives for each construct the activation rate according to each operation.
6. DMS-dependent optimization. The schema is restructured and denormalized, redundancy is added according to the specific characteristics of the DMS and of the computer environment in general. The files are defined. The schemas are evaluated on the basis of the statistics available and the best one is retained. It is then ready for a DDL generation.

3.4.6 Schema transformation

As already stated, the description of schema transformations will be developed in chapter 7. However, we can mention some of the most useful transformations in the logical design step.

- transformation of a N-ary relationship type into an entity type + N one-to-many relationship types;
- transformation of a set of attributes of an entity type into a new entity type + a relationship type;
- transformation of a relationship type into attributes + referential constraint;
- transformation of an entity type into attributes;
- replacing a compound attribute with its components;
- replacing a compound attribute with the concatenation of its components;
- replacing a multivalued attribute with several similar single-valued attributes;
- replacing a multivalued attribute with one single-valued attribute which contains the concatenation of its values;
- grouping attributes as a compound attribute;
- splitting an entity type into two entity types (+ a one-to-one relationship type) that share the attributes and roles of the former;

3.5 PHYSICAL DESIGN

3.5.1 Principles

During this step, the optimized DMS-compliant logical schema is translated into the DMS data description language (DDL). Some physical parameters can be given values at this level. They define technical characteristics such as physical allocation of data (e.g. logical/physical file mapping), page size, free space percentage for data and index pages,

buffer size, journaling management, clustering, etc. This DDL schema can be processed by the corresponding DMS processor in order to build the database.

The parameters are set according to efficiency needs. However, some settings can be induced, at least partially by higher-level specifications. In some relational DBMS, rows can be physically stored next to related rows (ORDER-LINE rows can be stored in the same page as their corresponding ORDER row), a storage schema that is called clustering. It is clear in this case, that this clustering supports the conceptual relationships between ORDER and ORDER-LINE entities.

3.5.2 Loss of conceptual information

It is important to note that in this phase, not all conceptual structures will be translated, even when a DMS-compliant equivalent structure has been included in the logical schema. There are two reasons for that state of things, namely weaknesses of the DMS and performance.

First of all, the DMS model and its technical constraints do not always offer constructs to express some logical constructs. For instance, a conceptual one-to-many relationship type will be translated into reference attributes plus a referential constraint into a COBOL- or relational-compliant schema. However, referential constraint is an unknown concept in COBOL files, and some relational DBMS still offer no way to translate it yet.

On the other hand, a complete translation of all the logical schema constructs is not always desired. For instance, implementing a referential constraint in the relational DBMS SYBASE is quite easy with a trigger. This feature, however, will be activated whenever a concerned row is inserted, deleted or updated. In an operational context in which strong validation procedures are enforced, and no direct access is allowed to the database, no invalid operation can be asked for, and therefore, the referential constraint will be checked without benefit. For performance purpose the designer can be tempted by ignoring the referential constraint and by not implementing it [9].

These observations have important consequences on the distribution of integrity control. Among the conceptual constructs that have been represented in the optimized DMS-compliant logical schema, a subset of them only are explicitly represented in the DDL text. The other part is either ignored (in which case the database is ill-defined and prone to quick degradation), or implemented by non-DMS resources.

Among the other means of implementing an integrity constraint, we can mention :

- pre-validation by specific filtering batch programs that select correct data to insert;

---

[9] this discussion can be criticized in the context of a strict DB design philosophy. Trade-offs opposing conceptual correctness and completeness on the one hand, and efficiency on the other hand often lead to unsatisfying solutions according to some criteria. Whatever the principles we choose to stick to when designing a database from scratch, the problem here is to understand the state of existing databases on the design of which we had no control.

- run-time validation by each application program; in this case, the responsibility to check the constraints is up to each programmer, a common but very dangerous situation;
- run-time validation by common access modules used by application programs (this point will be developed herebelow);
- run-time validation by the structure of the program; e.g. if entering a new order follows the designation of the customer, there is no need to verify that the CUST-NUMBER value in the ORDER row is correct;
- validation by the Man/Machine user interface (e.g. SQL-FORMS of ORACLE, which allows the definition of complex validation procedures for each screen field);
- post-validation by database scanning programs that detect and report errors periodically;
- validation by organizational constraints; e.g. if a constraint states that an order date must be a working date, and if the operators are not allowed to work but on working days, and if the order date is automatically set to the date of the order creation date, then there is no need to implement the checking of the constraint.

An example of run-time validation can be illustrated as follows (see figure 1.5) :

```
 fd F-CUST;
 record is CUSTOMER.
 01 CUSTOMER.
    02 CNUM pic X(14).
    02 CDATA pic X(57).

 fd F-ORD;
 record is ORDER.
 01 ORDER.
    02 ONUM pic 9(8).
    02 O-DATE pic X(12).
    02 CUST pic X(14).

working storage section
 01 CN pic X(14).
 01 C.
    02 CNUM pic X(14).
    02 filler pic X(57).
 01 O.
    02 ONUM pic 9(8).
    02 O-DATE pic X(12).
    02 CUST pic X(14).

procedure division (in pseudo-code)
    ...
    read CUSTOMER(CNUM = X) into C
    if found(C) then
          O.ONUM := ...
          O.DATE := ...
          O.CUST := C.CNUM
          write O into ORDER
    endif
    ...
```

Conclusion concerning the reverse engineering process

We can conclude from these facts that the DDL texts we can analyse will give a part only of the (transformed) conceptual structures that are (were) in the DMS-compliant logical schema. Important hints can be found for instance by analysing the procedural parts of the information system or the data entry forms. It is obvious too that these sources of information are far more complex to analyse than the DDL texts, whatever their intricacies.

3.6 USER'S VIEWS DEFINITION

A last activity in database design is to distribute views to the different users (endusers or programmers).

In most cases it is possible to proceed as follows :

1. a view is derived for each subsystem. Ideally, this view is a mere DMS translation of the normalized preliminary conceptual schema of that subsystem.
2. from that subsystem view, several reduced views are produced to fit the needs of specific users or applications.

An important way to implement the notion of view is through access modules. This technique is often used with DMS that don't offer powerful enough functionalities for defining views[10]. The topic will be developed in the next section.

3.7 DATABASE ASPECTS OF PROGRAM DESIGN

Despite the basic axiom of the database approach to application design, an axiom that states that data must be designed before, and independently of, the programs, there are important relations between the database and the programs at the operational level.

We shall describe four of them.

3.7.1 Relation with the data description

In modern DMS, the schema of the database is stored in a centralized location. In some cases (CODASYL, IMS), the schema is in a textual format that has been written only

[10] simple file systems are of course in that case. However, even the most sophisticated relational DBMS are not complete with that respect. Let's only mention the general weakness of the notion of relational view when update operations are concerned.

once, that has been processed by the DBMS compiler and that can be consulted at will (provided the source text has not been lost). The database itself contains a condensed (and generally unreadable) version of its schema. An application program need only mention the name of the database it wants to use. After that, the data structures are automatically known by the program.

In older DMS (COBOL), the schema is an integral part of each program that wants to access the data. Since each program is free to define its own view of the data, it is difficult to get a complete view of the schema. There is generally no complete schema, but only a collection of partial schemas. Quite naturally, these schemas are redundant (they describe the same data several times), complementary (no one describes all the data structures), and sometimes contradictory (some descriptions are not consistent with each other). A partial solution to this problem is to enforce (how ?) the following discipline : the data structures are described in a specific COBOL text. Every program which needs to access the files must include this text (with a COPY statement). If this rule is followed, the common description can be considered as the schema of the database.

Finally, in some environments (e.g. in VAX-VMS), a data dictionary contains the descriptions of the files and of the databases. The compiler offers facilities to allow a program to reference the data dictionary contents. In such a case, the schema is in the data dictionary.

3.7.2 Structure hiding and redefinition

The structure of a record type can be grossly defined in the database (e.g. with only one character-string attribute), while user programs have the responsibility to define more precisely its decomposition into distinct fields. In these cases, the DDL text gives no information about the fields. This information will be translated into the structure of the variables that will receive the data read from, or to be written in the database.

Such a common practice (e.g. with COBOL files or IMS databases) makes data structures difficult to observe since it can only be retrieved by following the data flow in the procedural statements of the programs. This difficulty is even increased when the receiving and emitting variables themselves are grossly decomposed as well.

Structure hiding can be sketched as follows :

```
Initial record structure
      01 CUSTOMER
         02 C-KEY  pic X(14)
            03 ZIP-CODE pic X(8)
            03 SER-NUM  pic 9(6)
         02 NAME     pic X(15)
         02 ADDRESS  pic X(30)
         02 ACCOUNT  pic 9(12)

Coded record structure
      01 CUSTOMER
          02 C-KEY  pic X(14)
          02 filler pic X(57)
```

3.7.3 Constraints in program structures

A program that makes use of a DMS must often follow restrictive rules as far as program structures are concerned.

The following restrictions can be encountered :

- all the database statements must be in the main program;
- recursivity is not allowed;
- only a limited number of files can be opened simultaneously;
- the database currency indicators (or cursors) cannot be transmitted as argument to a procedure;
- the database currency indicators (or cursors) are reset at termination of a transaction;
- two transactions cannot be embedded;
- an iterative access statement cannot be written inside another iterative access statement;
- a triggered database procedure cannot receive arguments from its triggering program.

These constraints make some otherwise simple programs really tricky. This consequence is important as far as reverse engineering is concerned.

3.7.4 Accessing data through access modules

A very common programming structure consists in hiding all the database statements into an interface module (subprogram or set of procedures + data structures). This practice gives a greater program independence according to several criteria [HAINAUT,86] :

- the syntax of the DMS language can change (version upgrade);
- the DMS can be replaced with another DMS based on the same model (one among ORACLE, DB2 and SYBASE);
- the DMS can be changed with another DMS based on another model (converting from CODASYL to relational);
- the logical schema can be modified (splitting a file into two partial files; merging two files into a single one; eliminating a derived attribute);
- the conceptual schema can be modified (new attributes, new entity types);
- etc.

Another great advantage of access modules is to complement the weaknesses of the DMS. For instance, referential integrity management can be centralized in an access module that control access to COBOL files.

Using access modules gives the application programs a high level of stability against a large range of perturbations (syntactic, technical, access, conceptual). When the database schema changes, at any level, modifying the internals of the access module(s) while keeping the same interface allows to keep the application programs unchanged.

The existence of access modules may make the reverse engineering of database easier since it implements a kind of centralized schema description.

3.8 DATABASE DESIGN PROCESSES AND STRATEGIES

The analysis that has been presented so far is fairly informal. In fact, we must distinguish processes from strategies, and define precisely the documents that are produced, as well as their relationships with the processes. Above all we need a more precise way of defining design methodologies. We will present a simple model to specify methodologies, and we will apply it to a more precise definition of database design.

A process is a closed activity that is aimed at solving a problem according to a limited set of criteria. A strategy is a way to carry out these processes in order to solve a more global problem. It is a temporal and decision-based arrangement of activations of processes.

A process has an external view and can have an internal view.

Through its **external view**, the process is perceived as a black box that transforms input products into output products according to specific criteria (generally called *requirements*).

This view is depicted in figures 3.1 and 3.2. It states that process $\texttt{PROCESS}_m$ basically is a transformation function such that :

$$\texttt{PRODUCT}_j = \texttt{PROCESS}_m(\texttt{PRODUCT}_i, \texttt{REQUIREMENTS}_k)$$

where $\texttt{PRODUCT}_i$ and $\texttt{PRODUCT}_j$ are set of products, and $\texttt{REQUIREMENTS}_k$ is a set of requirements.

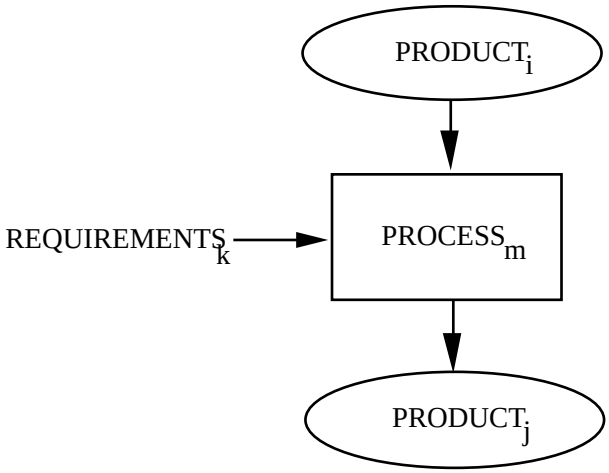


Figure 3.1 *- External (black box) view of a design process.*

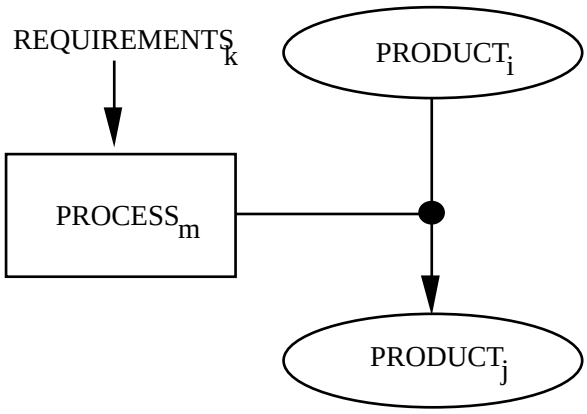


Figure 3.2 *- External (black box) view of a design process. An alternate presentation*

Through its **internal view**, the way the process is carried out is described. In general terms, this *way* will be called a strategy. It states how to produce the output products, i.e. with what procedure, with which reasonings, by activating which lower-level processes, with which internal products, etc. Figure 3.3 illustrates informally the internal view of a process. Any lower-level is in turn a process, and can be described through its external view, and, if it is not a primitive process, by its internal view.

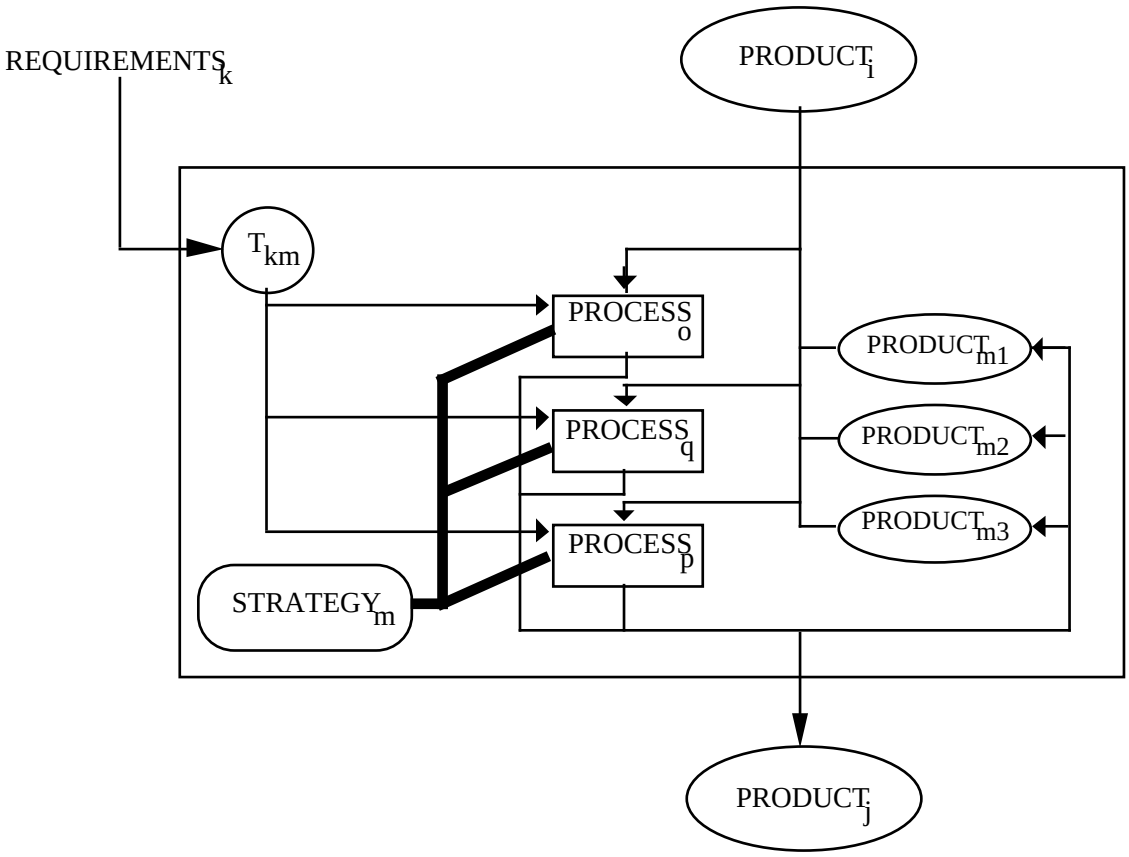


Figure 3.3 *- Internal (white box) view of a design process. The external requirements are be translated (T_{km}) into specific requirements, and distributed across several internal processes. Some internal products may appear. At least one process has* `PRODUCT`$_i$ *as input, and for each elementary product from* `PRODUCT`$_j$*, there is one process from which it is an output. The strategy specifies how the lower-level processes are enacted.*

Let us now use this model to define the structure of database design methodologies.

Figure 3.4 describes the external view of the global process. It shows that this process transforms some precise expression of user's requirements into four documents related to the database, namely its conceptual schema, the DBMS schema coded in the DBMS DDL, user's view coded in the DBMS DDL and in host language program sections or procedures that take in charge part of the conceptual schema that has not been translated in the DBMS schema.

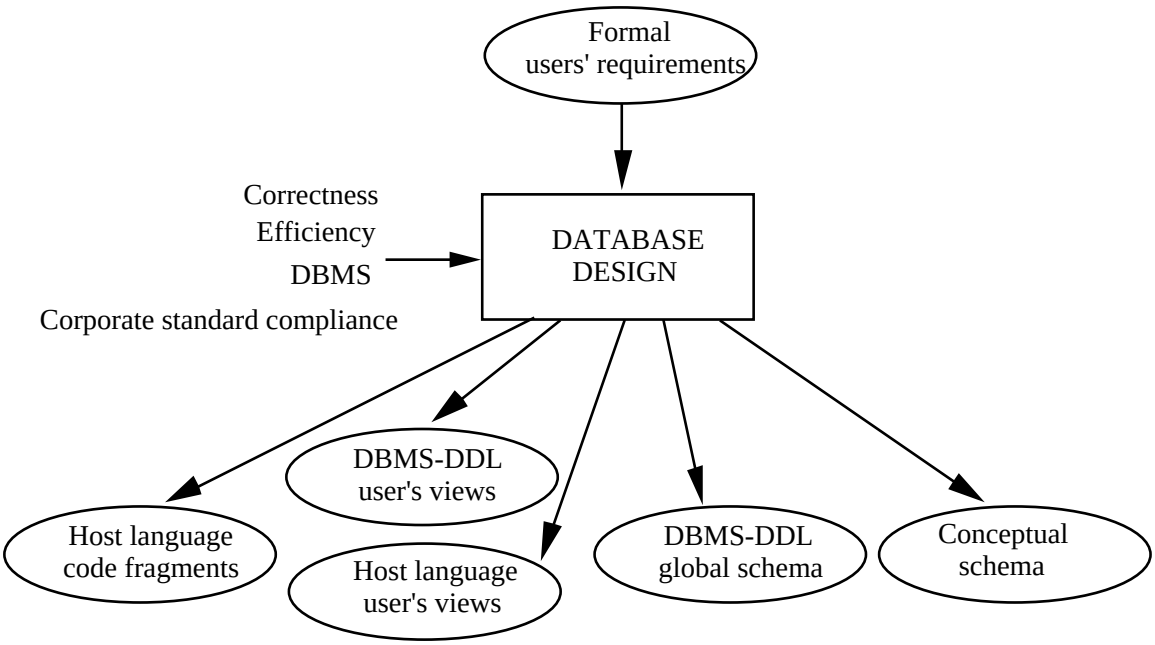


Figure 3.4 *- Global view of the database design process..*

Figure 3.5 shows one of the most popular strategy[11] for the **`DATABASE DESIGN`** process, i.e. its sequential decomposition into `CONCEPTUAL DESIGN`, `LOGICAL DESIGN` and `PHYSICAL DESIGN` [TEOREY,89] [BATINI,92]. The external requirements are translated into internal requirements and distributed among the latter processes. Regarding `CONCEPTUAL DESIGN` for instance, `Correctness` is understood as `Normalization`, `Efficiency` is translated into `Clarity` and `Minimality`, while `Corporate standards` should read `Modeling standards`. `DMS` (i.e. mentioning the target DMS) is not relevant here.

[11] This first example of strategy deserves some comments on the proposed design process model. Indeed, the specifications depicted in figure 4.3 can be interpreted in several ways. One of them is the **sequential/parallel** process execution scheme, according to which a process can be executed when all its input products are complete. For instance, `LOGICAL DESIGN` can be carried out only when `CONCEPTUAL DESIGN` is completed. According to the **pipelined** execution scheme, a process can be started when some parts of its input products are available. For instance, `LOGICAL DESIGN` can be carried out on those components of the conceptual schema that are available. The sequential/parallel scheme is generally implicit in database forward engineering, while the pipeline scheme is more familiar in traditional software engineering. The latter will also be a natural behaviour in large reverse engineering projects.

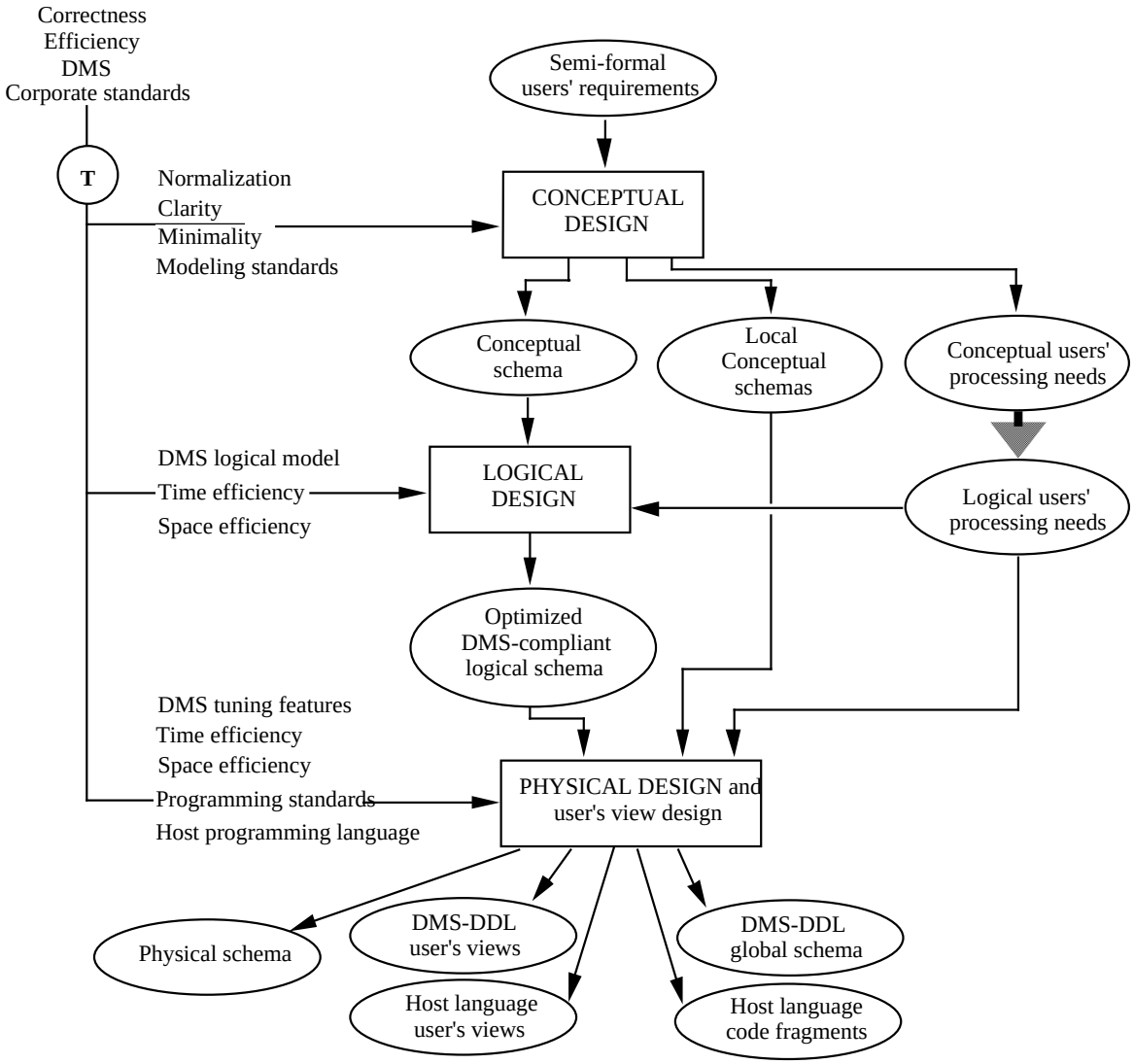


Figure 3.5 *- A typical strategy for* ***database design****.*

- The `CONCEPTUAL DESIGN` process transforms users' requirements[12] into `Local Conceptual schemas`, each dedicated to a subsystem of the organization, and into a `Conceptual schema`, that integrates the contents of the latter.
- The `LOGICAL DESIGN` process translates the `Conceptual schema` into an efficient schema according to the data model[13] of the target DMS (the so-called `Optimized DMS-compliant logical schema`).

---

[12] Users' requirements are so important in this domain that they have been considered as input products instead of as requirements.

[13] There is a large agreement on the fact that this model is that of the family of the DMS e.g. relational, CODASYL DBTG) rather than that of the DMS itself [BATINI,92]. This account for some genericity of this process. Taking into account the precise details of the specific DMS is generally considered as related

- The PHYSICAL DESIGN process transforms this schema into two computer code documents, namely the DMS-DDL global schema and the Host-language code fragments. The first document defines the data structures managed by the DMS, expressed in its DDL, while the second document implements, most often procedurally, the management of the structures (mainly integrity constraint) that have not been translated into DDL. This unfortunate splitting is due to the weaknesses of the DMS in expressing the contents of the DMS-compliant logical schema, and therefore of the conceptual schema. This process also yields the Physical schema that specifies the implementation techniques and parameter values chosen to get optimal performances. Users' views design has been presented as a part of Physical design. It consists in producing the views required by applications and end-users. According to the power of the DMS, these views will also be translated into DMS-DDL code and host-language code fragments (Host language user's views).

A secondary path concerns the processing needs deriving from user's requirements. These requirements are first expressed at the conceptual level, for instance as E-R queries. Then they are translated into the DMS query language or in some abstraction thereof (the Logical user's processing needs) , through a process that is not explicitly shown in the figure.

The **CONCEPTUAL DESIGN** process can be carried out according to the strategy suggested in figure 3.6. Its lower-level processes are described as follows.

- The user's requirements are decomposed and classified according to the organization subsystems, and according to their scope : structural and processing needs[14], new requirements and existing applications.
- REQUIREMENTS ANALYSIS transforms new user's requirements of each subsystem into a first-cut conceptual schema.
- DATA STRUCTURE REVERSE ENGINEERING transforms the DMS data structures of each subsystem into their underlying conceptual structure. Reverse engineering is now perceived as an integral part of forward engineering [BATINI,92], while, as will be observed later, several forward engineering processes are common with reverse engineering.
- CONCEPTUAL NORMALIZATION is intended to clean the conceptual schema of each subsystem to eliminate constructs that are considered as undesirable according to the chosen modeling standards. For instance, non-BCNF entity- or rel-types, multivalued compound attributes, structural redundancies, entity type with only one attribute, or without attribute, can be considered as unnormalized in some methodologies.

---

to physical design. In addition, this generic model is generally augmented by additional features that allows it to express all the structures and constraints of the conceptual schema.

[14] This strategy does not account for integrated approaches such as those based on the OO paradigm. Let us remember that our aim is to describe how *legacy systems* have been built. The reader interested by OO reverse engineering of old systems can be referred to [JACOBSON,91] for instance.

- `USERS VALIDATION` is the process that modifies the conceptual schema of each subsystem according to the evaluation made by the users. This evaluation can be conducted through, e.g. prototyping or natural language schema interpretation.
- `CONCEPTUAL INTEGRATION` produces a unique conceptual schema that contains the semantics of all the local conceptual schemas.

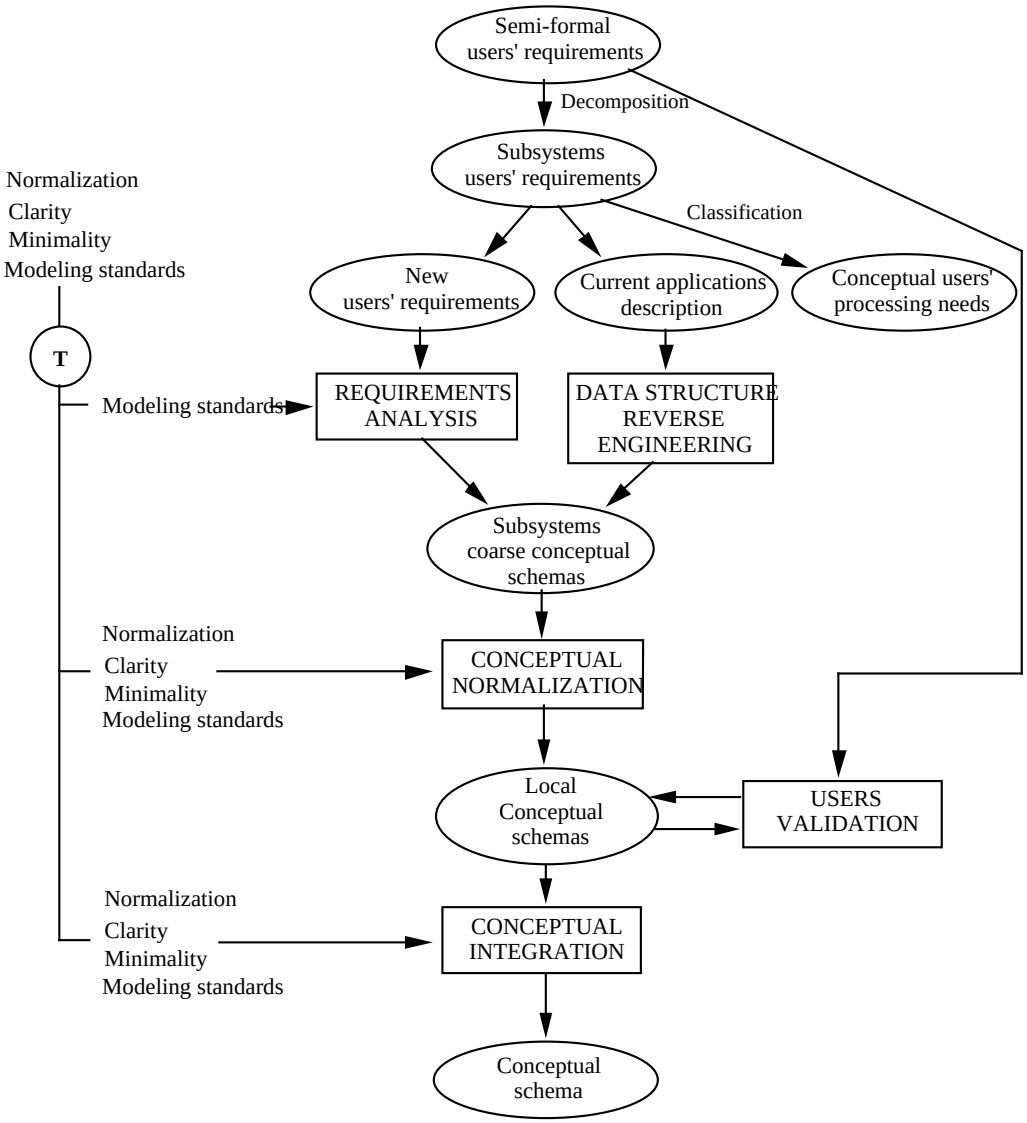


Figure 3.6 *- A typical strategy for* ***database conceptual design****.*

The **`LOGICAL DESIGN`** process can be developed according to the linear strategy sketched in figure 3.7. It implies performing up to four lower-level processes :

- `SCHEMA SIMPLIFICATION`, through which the schema can be transformed into a more simple model (e.g. Bachman's model), better suited for optimization reasonings; for instance, n-ary rel-types are transformed into binary ones, multivalued attributes are reduced to single-valued ones, IS-A links are transformed into one-to-one rel-types.

- `DMS-independent OPTIMISATION`, through which the schema can be first optimized according to general rules that can apply whatever the target DMS. More generally, the schema can be restructured according to design requirements concerning access time, distribution, data volume, availability, etc. Schema transformations such as vertical/horizontal structure splitting or merging, denormalization or structural redundancy are commonly used to satisfy these requirements.
- `DMS TRANSLATION`, that transforms the schema into structures that comply with the target DMS model. For instance, for standard files (or relational DB) many-to-many rel-types are transformed into record types (tables) while many-to-one rel-types are transformed into reference fields (foreign key). In a CODASYL schema, a secondary identifier is represented by an indexed singular set. In a TOTAL or IMAGE database, a one-to-many rel-type between two major entity types is translated into a *variable entry* record type.

  On the other hand, due to the limited semantic expressiveness of older (and even current) DMS, this translation is seldom complete. It produces two subsets of specifications : the first one being strictly DMS-compliant while the second one includes all the specifications that cannot be taken in charge by the DMS. For instance, referential constraints cannot be processed by standard file managers. In principle, the union of these subsets includes the conceptual specifications.
- `DMS-dependent OPTIMISATION`, that performs further optimization transformations according to specific rules that depend on the DMS.

All these processes behave as semantics-preserving transformations.

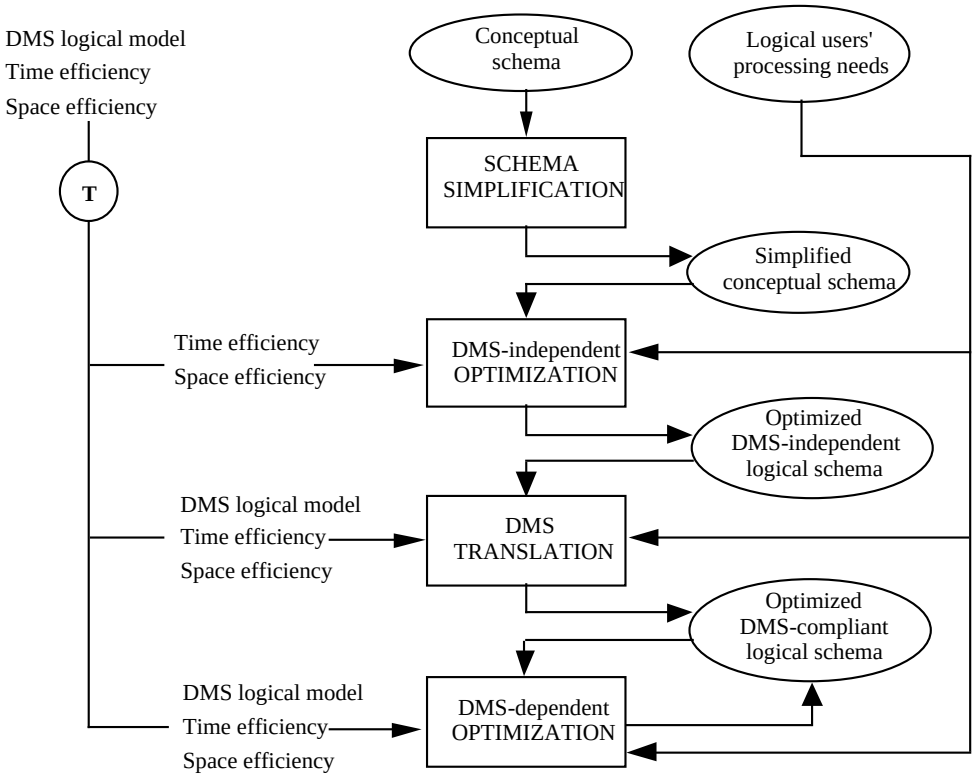


Figure 3.7 *- A typical strategy for* ***database logical design****.*

A simple but realistic strategy for the **PHYSICAL DESIGN** process is proposed in figure 3.8. The first branch is dedicated to the production of the executable schema, the second one is aimed at generating the user's views, while the third one is concerned with physical database tuning. The four major lower-level processes are described as follows.

- DMS-DDL/Host CODING translates the first subset of specifications (those that collect strictly DMS-compliant structures) into the DDL of the DMS. The discarded specifications are translated[15] into languages, systems and procedures that are out of control of the DMS (e.g. host language, user interface manager, human procedures, etc). Let's observe that the DMS-DDL schema is not always materialized. Many standard file managers, for instance, doesn't offer any central way to store the description of the files, e.g. in a data dictionary.
- User's views DERIVATION defines the subset of the schema that concerns each application and end-user category. The Local conceptual schemas are a major input for this process.
- User's views CODING expresses these views into executable code. According to the DMS and programming standards and habits, this code may consist in two sections. The first section is made of a DDL text that translates some of the structures of the view. The second section expresses the structures discarded in the DDL text in host-language code. A frequent example consists in declaring the fields of a COBOL record type, or of a compound field, as an unstructured anonymous filler field, and in introducing in the programs a working variable in which records can be stored, and which is given the origin field decompositio. This practice, called **structure hiding**, is frequent in IMS and CODASYL applications too and has been observed in SQL applications as well.

[15] or too often ignored, in such a way that they are definitively lost !

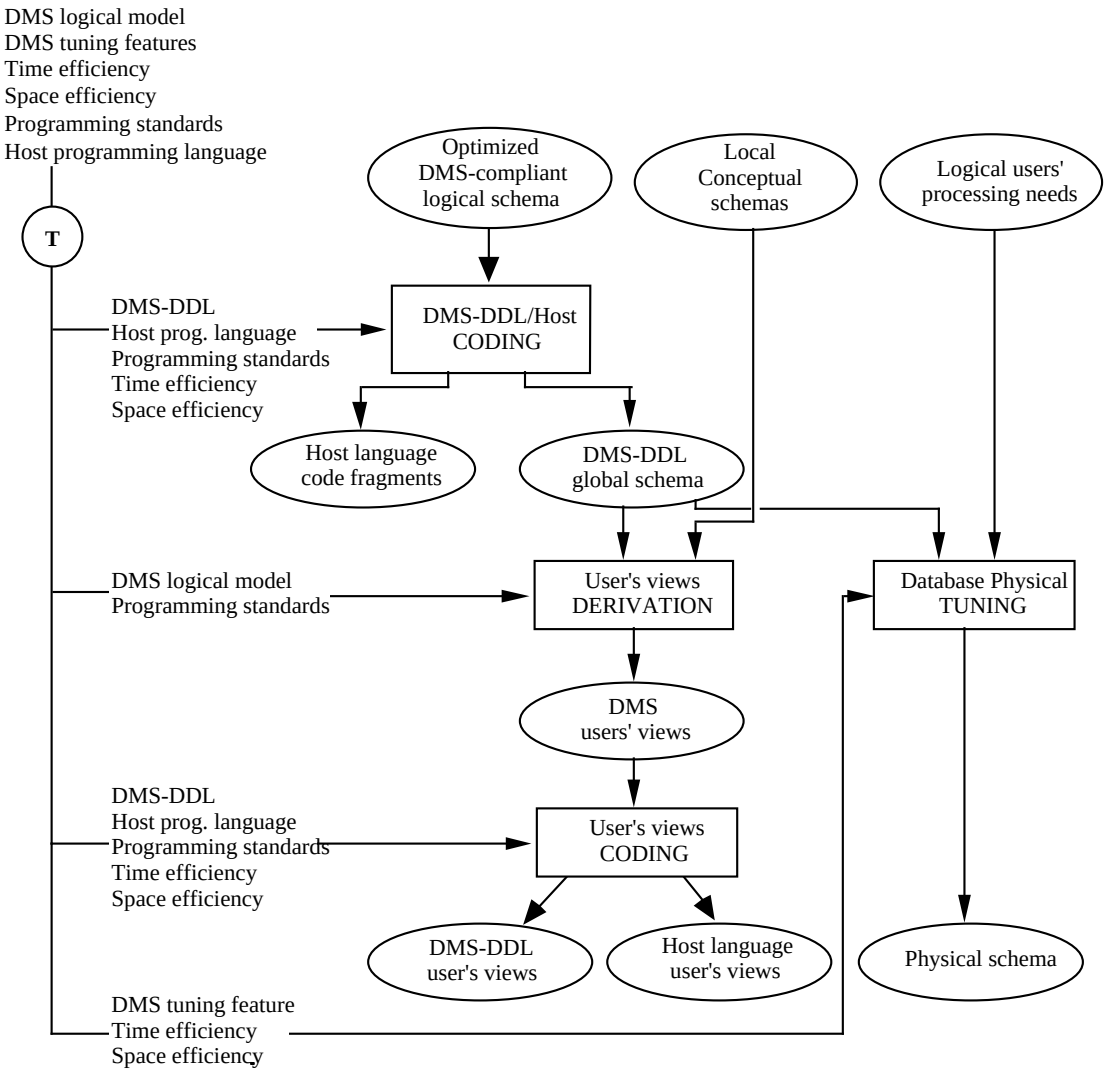


Figure 3.8 *- A typical strategy for* ***database physical design*** *and* ***user's views derivation****.*

- `Database Physical TUNING` defines the implementation structures and the parameter settings in order to make the database as efficient as possible according to the user's processing needs. These specifications form the `Physical schema`. According to the DMS, it consists in extending the DDL schema, and/or in a specific document. For instance, the physical schema will include the specification of indexes, physical file assignment, device/media assignment, record type space mapping, page size, free space percentage, clusters, storage modes, access modes, buffer size and management, etc.

Most of these processes can be further refined, directly or indirectly, into lower-level processes that are primitive schema transformations as defined in section 3[16]. As an illustration, figure 3.9 proposes a simple strategy based on semantics-preserving

[16] This transformational top-down refinement model is explicitly advocated in [BATINI,92], [HAINAUT,81], [HAINAUT,92a], [ROSENTHAL,88] for instance.

transformations (that reduces to a strict algorithm) for the DMS TRANSLATION process instantiated for standard relational structures.

```
1. for each non-functional rel-type R, do : transform R into a rel-
   type;
2. while compound or multivalued attributes exist, do :
   for each attribute A that is both single-valued and compound,
   and that depends directly on an entity type, do :
      replace A by its components;
   for each attribute A that is multivalued, and that depends
   directly on an entity type, do : transform A into an entity
   type;
3. until no rel-type can be transformed, do :
   for each (functional) rel-type R, do :
      if R can be transformed, then transform R into reference
      attributes;
4. until no rel-types remain, repeat :
   for each rel-type R, do :
      add an artificial attribute A to E, and make A the
   identifier of E;
      do 3;
5. make each identifier and each foreign key an access key;
6. remove each access key that is a prefix of another access key;
```

Figure 3.9 *- A typical strategy for Relational* ***DMS TRANSLATION****.*

Chapter 4

OBJECTIVES OF DATABASE REVERSE ENGINEERING

This chapter presents briefly the global objectives of reverse engineering activities at the organization level. With this respect, this objective is clearly at the strategic level. Therefore, its contents will be further developed in chapter 14.

4.1 INTRODUCTION

Technically speaking, the reverse engineering is a domain of the software engineering that consists in analysing existing, possibly old, applications in order to recover their logical and functional structure. In many cases, the only source of information on their functions and on their structure is often the COBOL or PL/1 text of the programs.

A subset of the reverse engineering process consists in concentrating on the file and database structures used by the applications. Recovering these structures is central since they constitute the interface between program units and they incorporate in a very concise form an important part of the knowledge about the structure and the rules governing the organization (the database is an image of this organization). Understanding these structures will ease considerably the reverse engineering of the application programs. In addition, these structures are more stable than the programs themselves.

We shall analyse in which global enterprise activities database reverse engineering can be conducted. Obviously, these activities are not strictly distinct, some of them being specializations deriving from a more global objective.

4.2 APPLICATION REENGINEERING

Rewriting and restructuring an old application is the most obvious example of software engineering activities that need reverse engineering. The very objective is to obtain a better version (in terms of documentation, structure, maintainability, performances, up-to-date resources, etc) of the existing application.

4.3 APPLICATION REDOCUMENTATION

The application is left unchanged. However, the documentation is lost or outdated. The objective is to actualize the documentation.

4.4 APPLICATION MAINTENANCE

The application must be slightly modified either to correct bugs, to modify some functionalities or to adapt it to new versions of resources (such as screen or data managers). However, the documentation is lost or outdated. The organization considers that a correct documentation is a necessary condition to program maintenance and ask for rebuilding it from the available sources.

4.5 APPLICATION CONVERSION (between DMS)

This fact is easy to demonstrate. A direct translation from DMS1 to DMS2 consists in expressing in DMS2 the optimized, DMS1-compliant logical schema of the current database. Therefore, the final schema will include inherited DMS1-dependent optimization constructs that are not suited for DMS2.later. This conversion cannot be carried out without a complete and consistent conceptual schema of the old database. Direct translation from one DMS to another generally leads to poor structure and performance[1].

[1] This fact is easy to demonstrate. A direct translation from DMS1 to DMS2 would consist in expressing in DMS2 the optimized, DMS1-compliant logical schema of the current database. Therefore, the final schema will inherit DMS1-dependent optimization constructs that are not suited for DMS2.

4.6 APPLICATION EXTENSION

The new version will include new functionalities, and particularly new database objects. A consistent extension needs a complete and consistent documentation (particularly the database conceptual schema) of the current version of the application.

4.7 APPLICATION INTEGRATION

The integration can be carried out according to several architectures sketched herebelow.

1. Several, independent applications are merged into a single one. One of the most difficult problems is to integrate the databases of the applications into a single one.
2. The database remain unchanged, but their management is synchronized[2]. For instance, data duplicated across several databases are updated simultaneously. This synchronization can be enforced either by rewriting some parts of the applications, or by interfacing the databases with a synchronizer which dispatches in real-time the updates toward the concerned databases, or by an a posteriori program that propagates the updates of a database to the other ones. This architecture requires a common, consistent view of the concerned databases.

4.8 APPLICATION DEVELOPMENT

As already quoted, application design, and particularly database design, uses a great variety of information sources, including analysis of current applications, as far as they have some commonalities with the application in project. Therefore, retrieving an up-to-date description of the current application will often need some kind of reverse engineering.

4.9 DATA ADMINISTRATION SUPPORT

Data administration is a vital department in organizations using large and complex files and databases. The very objective of the data administration is to register, maintain and distribute the description of all the relevant data of the organization. Ideally, these descriptions are stored into a unique, specific database called Data Dictionary (DD), or Information Resource Dictionary (IRD). The Data Dictionary contains the definition of the

---

[2] This problem is known in the literature as a multi-base or federated database configuration, in which several, possibly overlapping, databases coexist, though they may be managed by different DBMS on different sites

data (files, record types, relationships, tables, domains) but also of their usage (users, programs, update rate, etc) and of their environment (location, responsible, etc).

One of the biggest problems a DA encounters is that it has been developed far after the data have been defined and used. Therefore, most data currently used have no correct description (if any). Building a complete and consistent map of the enterprise data typically is a database reverse engineering problem.

Chapter 5

GROSS ARCHITECTURE OF REVERSE ENGINEERING

This chapter analyses the reverse engineering process as the converse of the forward engineering activities. It derives conclusions concerning the problems to be solved, a general organization for the reverse engineering and concerning general issues such as the sources of information and the mapping between the initial and the final data structure descriptions. The suggested procedure is then criticized.

5.1 REVERSE ENGINERING AS THE REVERSE OF FORWARD ENGINEERING

The first idea that comes to mind when considering reverse engineering, is that this process can be considered as forward engineering carried out the reverse way. Starting with the physical schema of the database, e.g. under DDL text format, we have to find out the origin conceptual schema, whether it ever existed or not. Let's try to push the comparison a bit further by considering that each step of the reverse engineering process is exactly the reverse of a database design activity.

In order to get some concrete ideas about the characteristics of the reverse activities, let's recall the main database design activities, and try to imagine, for each of them, the corresponding reverse engineering activity. These steps are considered in the reverse order, from physical to conceptual.

5.1.1 User's views definition

Considering this step can be useless or, on the contrary, very important depending on the DMS.

In **true DBMS**, user's views are derived from a unique DMS-compliant logical schema that generally is available under its DDL translation or as data dictionary contents. Knowing about these views may be of no interest. However, some systems offer different structural definitions in user's views. For instance, some CODASYL systems allow the definition of compound and multivalued fields in subschemas (user's views), but not in the schema (the logical DBMS schema). In such cases, user's views can bring richer structures than the global logical schema. In relational DBMS, view can give useful information through the names and agregates that the views are made of. In addition, the query that defines a view can make use of joins that suggest foreign keys.

In **simpler DMS**, such as standard file systems (supported by no data dictionary), the only formal description that is available is a collection of partial views spread, duplicated, and sometimes strongly hidden, into the application programs. User's views are the major source of information. If the number of programs is large, the same view, or parts of it, can be duplicated a large number of times, each instance bringing no further information (this is a kind of schema redundancy).

We can conclude from this analysis that :

- analysing user's views is of no interest for source DMS for which the global schema is available and complete. It can be useful when views are allowed to bring new information. On the contrary, it is a major activity in standard file systems.
- a first RE activity is to find all the sources that may include a user's view (such as programs).
- a second RE activity is to find out, in each source, the user's view that is used. Since this activity is related to physical design, it will be discussed in the next section.
- there can be a large number of identical or similar user's views.
- therefore, a third RE activity is to reduce the number of identical or similar user's views.

5.1.2 Physical design

This step translates the optimized DMS-dependent logical schema into a DMS DDL schema. Some conceptual informations cannot be translated that way. They are implemented through various different ways. The most important of them is through the procedural code of the programs.

The reverse engineering activities deriving from this step can concern either the whole schema, when it is available, or each user's view, when this whole schema is spread across several sources. Reversing the physical design step consists in retrieving the DMS-dependent logical schema from its various implementations (DDL, procedural, others).

The result is either a single logical schema or a collection of partial logical schemas. In some situations, this step is the most difficult and complex reverse eng

5.1.2.1 Physical parameters setting

Most physical parameters have no value for reverse engineering.

However some specifications can be useful, such as

- clustering definition, that can suggest foreign keys in simple relational databases,
- logical file / physical file assignment, that can give additional names to files, and that can relate several logical files that are assigned to the same physical file.

5.1.2.2 Translation of the DMS-compliant logical schema into the target DDL

This process corresponds to an important reverse engineering activity. If the complete DDL expression is available, then the task is easy. However, for some DMS such as standard file managers, the description can be quite difficult to find out. Indeed, the explicit information found in the source texts is rather gross and gives few details (e.g. it gives the names of the record types but no information about the fields). In that case, a careful and tedious analysis of other parts of the source texts, parti

It is important to note that this difficulty does not result from the limits of the DMS as far as data structure expression is concerned, but rather from programming practices used by some programmers.

5.1.2.3 Translation of discarded conceptual information into non-DMS expressions

The conceptual information that cannot be translated into the DDL of the DMS must be implemented in another way. Generally, there exist somewhere procedural sections that are in charge of ensuring the data integrity according to this lost information. Let's remind the case of referential integrity in standard file systems. In some cases, the procedural section is centralized (access modules or by COPY or INCLUDE statements). In some other cases, these statements are spread into all the programs.

Conclusion

This analysis leads to emphasizing the following reverse engineering activities :

- analysing the physical parameters for hints about conceptual structures or view organization,

- analysing the source texts in order to find out the complete description of the translated logical structures; this activity can be heavily complicated by specific programming practices that tend to hide these descriptions.
- analysing the non-DMS part of the source text in order to find out additional information on the logical structures that have not been translated into the DMS DDL.

5.1.3 Logical design

Through this step, a global, normalized conceptual schema is transformed into another schema that preserves its conceptual contents, that is compliant with the target DMS and that is optimized. A reverse engineering activity can easily be defined as the reverse of this step : to retrieve the underlying conceptual schema of an optimized DMS-compliant logical schema.

5.1.3.1 Name translation

If the names have been translated through systematic rules, it can be fairly easy to find the origin conceptual names. Otherwise, the process can be more complex.

5.1.3.2 Translation of the conceptual schema into a DMS-compliant logical schema

As stated and discussed earlier, there is no one-to-one relation between conceptual schemas and their DMS-compliant translations. Therefore, given a DMS-compliant logical schema, there are several possible conceptual schemas it may have been derived from. When the database has been correctly implemented, the direct translation process has made use of semantics-preserving transformations[1]. Such a transformation T has an inverse transformation T' that can produce the origin schema when applied to the schema transformed by T. In most cases, the set of transformations is rather limited.

The reverse process raises two problems : to detect constructs that derive from a conceptual construct, and to determine what transformation T has been used when the database was built[2]. This process can profit from the knowledge of the most usual practices in database implementation in the concerned DMS : there are few ways to replace a many-to-many relationship type or a multivalued attribute in relational databases.

---

[1] some non-semantics-preserving transformations can be used. In most cases, however, the data management procedures will add logic such that the semantics is preserved. Transformation of a multi-valued attribute into a single-valued attribute is such a case.

[2] in many situations, no design methodology has been used at all. However, it is generally possible to rewrite a possible history of the design of the database in concern. All the discussions of this chapter refer to this virtual design.

Even in richer DMS (COBOL is one of them), the set of possible ways to transform non-compliant constructs is not so large.

Nevertheless, finding somewhat odd (though correct) translations may occur more than once.

Due to the structural constraints of the DMS, there can be some correlations between conceptual constructs and access constructs. For instance, some DMS require that every identifier is an access key as well. On the other hand, access paths (that are supported by relationship types) are converted into access keys in relational and standard file systems. Therefore, an access key can be an evidence for a reference attribute, and therefore for a transformed relationship type.

In conclusion, an access construct can give some information on the conceptual structures of the database.

5.1.3.3 Optimization of the logical schema

Optimizing a logical (either DMS-independent or DMS-compliant) schema is generally done by restructuring, denormalizing and adding structural redundancies. Restructuration is generally a semantics-preserving process. Therefore, as far as reverse engineering is concerned, the conclusions can be the same as those of the analysis of the translation of the conceptual schema into a binary schema (see below). The other two techniques can be reverse engineered provided the redundancies have been detected, which is the very problem at this level. Structures resulting from denormalization can be cleaned easily thanks to E-R normalization rules. Structural redundancies are simply eliminated.

5.1.3.4 Quantification and access mechanisms

Once the presence of access mechanisms has been analysed for hints about the possible conceptual constructs they support, they can be disregarded. At first glance, no major information can be derived from the quantification of the data[3].

5.1.3.5 Translation of the conceptual schema into a binary schema

In the context of reverse engineering, this step relates to the variety of semantically-equivalent conceptual schemas. The binary schema is only a version that can be accepted as such or further translated if needed. The only criteria that are effective at this level are

[3] some minor inferences can be thought of, such as the following : if an entity type has exactly 10 instances, and we have found 7 different values for the value-set of its identifier, we can conclude that we lack exactly 3 values so far. Such reasonings will be ignored.

the readableness, the conciseness and standard-compliance (each organization can enforce proper standards as far as the conceptual formalism allowed is concerned).

We can conclude from the analysis of the logical design step that the following reverse enginering activities are needed :

- name translation.
- detecting transformed constructs.
- semantics-preserving transformations.
- data redundancy detection.
- detecting unnormalized constructs.
- normalization transformations.
- schema updating.

Note : The discusion above is about the conceptual reconstruction of one logical schema. It is clear that it is valid whatever the schema, be it complete or partial. In the latter case, the result is the conceptual schema of a user's view that must be further integrated. The integration process can be undertaken between logical schemas of any level, even between optimized DMS-compliant logical schemas. However, this process must take into account not only conceptual structures but access structures and optimizing constructs (including redundancies and unnormalized structures) as well.

Let's propose another consideration : we have seen in section 3.6 that there can be a strong dependency between a user's view and the conceptual schema of the subsystem the user belongs to. Therefore, it can be easier in some cases to find the partial conceptual schema from which the user's view derives, then to integrate these partial schemas. On the other hand, processing a large number of (near-)identical user's views can be quite cumbersome, suggesting early simplification instead. The discussion of both approaches will be developed in section 14.7.

This discussion leads to a last conclusion : there can be a need for logical view integration activities at this level.

5.1.4 Conceptual design

At first glance, the final goal of the conceptual step is the same as that of the whole reverse engineering process : producing a good-quality conceptual schema. We could conclude that reversing this step is of no interest, and that nothing useful can be expected from examining it. This conclusion is not quite true. Indeed, producing a correct conceptual schema by normalizing and integrating a collection of partial schemas is a goal that is shared by both processes. Therefore, many aspects and activities of the conceptual design step can be adopted, **without being reversed**, in a reverse engineering process.

5.1.4.1 Decomposition of the application domain

Ideally, the applications for which the reverse engineering process will be carried out represent gross organizational functions, and are therefore related to major subsystems of the application domain. Besides this observation, no major information can be derived from this activity.

5.1.4.2 Building the preliminary conceptual schema for each subsystem

In problems based on standard file applications, the result from the previous reverse engineering activities can be a collection of partial conceptual schemas.

5.1.4.3 Normalization of the preliminary conceptual schemas

This problem has been partly discussed above. Constructs that lack generality, that are too complex or unclear can be transformed by semantics-preserving transformations. This process is partly dependent on specific methodological standards that state what should be an acceptable conceptual schema.

5.1.4.4 Integration of the preliminary conceptual schemas

In contexts and strategies that lead to a collection of partial conceptual schemas, the integration process is a major activity.

5.1.4.5 Validation of the global conceptual schema

This step is mentioned to recall that reverse enginering from source texts alone cannot produce acceptable results unless validation from concerned users is asked for. This validation may induce semantic enrichment of the schema.

From the analysis of the conceptual design step we can quote the following reverse enginering activities :

- detecting unnormalized constructs.
- normalization transformations.
- semantics-preserving transformations.
- View integration.
- Semantic enrichment.

5.2 PROPOSAL FOR A GENERAL MULTI-LEVEL STRATEGY

5.2.1 Introduction

The previous analysis of the database design activities from the reverse engineering viewpoint suggests us a framework according to which the main problems

- can be localized precisely, and tackled in specific processes,
- can be solved through a transformational approach.

Both this analysis and the first on-field experiments suggest that the activities can be organized according to two major processes. The first process leads to the elicitation of the source data structures from the multiple source texts (**data structure extraction**), while the second process searches these data structures for the underlying conceptual structures (**data structure conceptualization**).

According to the forward engineering design model sketched in chapter 3, the data structure extraction process can be seen as the reverse of the physical design phase and of the user's views definition. Its main output is the *optimized DMS-compliant schema*.

The data structure conceptualization process is dedicated to the transformation of the optimized, DMS-compliant schema into a conceptual schema.

The proposals of this section concentrate essentially on the data structure descriptions that can be found in operational source texts (e.g. programs).

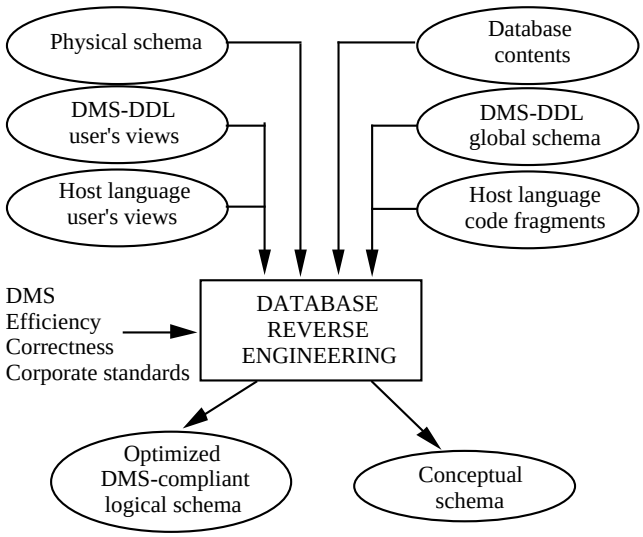


Figure 5.1 *- Overview of a theoretical reverse engineering process*

However, as could be expected, this view is highly idealistic. Before going into further details, let us examine some objections.

- *Most current databases have been built without any methodological concern. Even computer-aided database design uses formal approaches that are much simpler than the model presented hereabove.* That's true. However, one can admit that there exist one or several *designs*[4] that could have lead to the operational database to be reversed. The problem is to discover one of these possible designs.
- *In some situations, e.g. when processing standard file structures, there exists no DMS-DDL global schema.* Ideally, this schema should be available. When there is no way to store it in a systematic way, it has been "recorded" on paper, it's to say that it is now obsolete or even lost. The suggestion is to recover the unknown gobal schema through integration of the available users' views.
- *All the source code texts are not always available. Some design practices lead to discard semantic specifications such as non-DMS integrity constraints.* This means that some structural information cannot be recovered, and therefore will be lost. A final schema will be found, but may be uncomplete. A posteriori checking will be essential.
- *Some information are deeply burried into the source code texts, and are almost impossible to find out and to interprete.* Tricky heuristics, based on specific program patterns, must be used to analyse the source code. Computer support is required to parse huge volume of source texts in order to detect these patterns.
- *Errors can be found in the operational source code texts.* These errors have a chance to be discovered during the reverse engineering process since the physical structures will be reformalized and will be evaluated and criticized at various levels.
- *A source construct may be transformed into different target constructs, and different source constructs may be transformed into the same target construct. Determining which forward transformation has been applied, and therefore which was the source construct is not always an obvious task.* Experiments prove that choosing the adequate transformation basically is a knowledge-driven process that cannot be carried out fully automatically, except in simple and academic situations. On the other hand, if two possible abstract constructs can be found as representations of a single physical construct, this means that these constructs are equivalent, and that both are valid. Chosing one of them is a matter of methodological standard. A complete reverse engineering approache must take this into account.
- *It is impossible to formalize all the possible design practices and habits, particularly when the applications have evolved.* The proposition is based on the *non-perversity* axiom, that postulates that the developer have worked honestly, and with the aim to make the system work properly, without unnecessary complexity. This does not mean that the result is clear, clean and well structured. With this respect, we can consider that all the design processes that have been identified could have been carrie out, either implicitly or explicitly.
- *Some of the implicit or explicit requirements that have driven the design are unknown.* This makes eliciting the design choices more difficult. For instance, ignoring whether

---

[4] Let us call ***design*** the history of the building of the database. A design is a sequence of processes and a collection of products, together with the rationales for having executed these processes (the justification of design choices).

space optimization or time optimization were of highest priority can make interpretation of the physical schema more difficult. However, studying the context (software, hardware, developer's skill, organizational constraints) in which the application has been developed will generally give essential hints on what were the most important requirements.

- *The strategies that have been carried out are unknown.* This is not a real problem, since we have just to imagine a possible strategy that satisfies the same requirements (if any), and that can lead to the same result.
- *The objective of reverse engineering can also be to recover the requirements.* Let us observe that this concerns non-fonctional requirements only. Indeed, recovering users' requirements, expressed into a conceptual schema, is the primary aim of DBRE. Though this perspective will not be discussed in this document, the framework may prove adequate for non-functional requirements as well.

Let us now present the two main database reverse engineering processes, namely *Data Structure Extraction* and *Data Structure Conceptualization*.

5.2.2 The data structure extraction process

Through this step, the source texts are analysed in order to retrieve the optimized DMS-compliant logical schema. This step is mainly related to the physical design activities of database design.

The major activities of the data structure extraction step are the following.

- Collect the relevant information sources
- Analyse the physical parameters for hints about conceptual structures or view organization
- Analyse each source text in order to find out the DMS implementation of the logical data structures; some descriptions are explicit in the texts while others can only be elicited through a complex analysis of procedural parts;
- Analyse the source text in order to find out additional information on the logical data structures that have not been translated into the DMS DDL.
- If the source texts provide multiple views for the same data, and if these views induce too much redundancy, reduce this redundancy. This activity may need name processing.
- Build the relationships between the programs and the data.
- Keep trace on how and where each logical data structure is physically defined and used.

The results of the data structure extraction step are the following.

- A set of (at least one) DMS-compliant logical schemas.

Each schema is made up of entity types, relationship types (if the DMS can represent them explicitly), attributes and domains, identifiers, access keys and access paths, sequence ordering, entity collections (i.e. files) and additional integrity constraints. In addition, each object will be given its physical origin(s). For some objects, a semantic interpretation can be proposed.

- The description of the programs that have been analysed and of their relationships with the logical schemas.

This process is developed in chapter 11.

5.2.3 The data structure conceptualization process

This step consists in constructing a conceptual schema from the optimized DMS-compliant logical schema obtained in the data structure extraction step. It is mainly related to the logical design activities of database design. However, the analysis developed in section 5.1 has shown that some important problems in this step are related to conceptual design as well. Such is the case for schema integration and normalization.

The major activities of the data structure conceptualization step are the following.

- process names in order to make them more conceptual
- integrate the multiple views. This can be done at different stages of schema processing, from the DMS-compliant version of the schemas (early integration) until their conceptualized version (late integration).
- Find redundant structures and eliminate them
- Find unnormalized structures and transform them
- Keep trace of all the logical schema origin of the objects of the conceptual schema
- Add new semantic constructs (semantic enrichment).

The results of the data structure conceptualization step are the following

- A unique conceptual schema. This schema is made up of entity types, relationship types, attributes and domains, identifiers, additional integrity constraints and semantic interpretation. In addition, each object will be given its physical origin(s).
- The relationships between the conceptual schema and the DMS-compliant logical schemas.

This process is developed in chapter 12.

5.2.4 Other activities

Final processing of the schema can occur in order to give it such characteristics as readableness, conciseness, generality and methodological standard compliance. This processing consists in semantics-preserving restructuration of the schema, and relates to activities that are typical of conceptual design.

5.3 THE MAIN PROBLEMS IN DATABASE REVERSE ENGINEERING

Whatever the way a reverse engineering process is conducted, the same general problems will be encountered sooner or later. They will be analysed in the next chapters.

5.3.1 Name analysis

Names are important information about the concepts they denote. Ideally, the name of a data object must evoque, and even identify a unique concept belonging to the application domain. In practice, however, things are not so simple.

Here are some identified sources of problems.

- **role** : the same concept can be used in different places with different roles. The naming conventions may reflect that.
- **syntactic variant** : a name can be written with different spelling, including wrong ones.
- **language translation** : the use of more than one (natural) language may account for naming variety.
- **context augmentation** : a name may include some information on the context in which it is defined or used.
- **undisciplined naming** : not following naming conventions makes name interpretation harder.
- **excessive abbreviating** : when a name is made up of several (generally hierarchical) parts, each of them generally is abbreviated. Too short abbreviations are difficult to interprete.
- **synonyms** : the same concept can be denoted by several names.

This point will be developed further in chapter 6.

5.3.2 Data redundancy

The problem is twofold : detecting data redundancies and reducing them. A data redundancy exists when the data structures allows multiple representations of the same real-world fact. In short, there may exist more that one data item that represents the same fact. As described in section 2.3.11.6, redundancy is not a symmetrical concept : B is redundant with A doesn't mean that A is redundant with B. Construct B, not A, must be discarded. Redundancies are undesirable when trying to reach a conceptual description of the data.

There are two main sources of redundancies in a database schema : structural redundancy and unnormalized structures (see 3.4.3).

- Structural redundancy consists in data structures the instances of which can always be derived from other data. It is materialized by an attribute, an entity type or a relationship type that can be eliminated.
- An unnormalized structure cannot be detected by a simple examination of the entity types, relationship types and attributes of the schema. They are induced by undesirable dependencies (mainly functional and multivalued) among the attributes and the roles of an entity type or of a relationship type. These dependencies lead to several representation and updating abnormalities. Such structures can be eliminated by decomposing the entity type or relationship type in concern.

This point will be developed further in chapter 9.

5.3.3 The multiple view problem

This concept relates to the fact that the same data have been given more than one description. Such a situation occurs in systems that allows multiple views (or schemas) to be defined on the same database. Basically, this phenomenon is not a problem in database reverse engineering, except when there is no global schema available (whatever its level) that describes the database. In DMS such as standard file systems (when no data dictionary is used), the data are described in each program that makes use of them. Since the needs of each of these programs can be different, the ways they access these data can be different as well. Analysing a program from the data description viewpoint gives a partial view of the database only. Quite naturally, these multiple views are not only complementary, but also overlapping, i.e. redundant. Indeed, several programs that need the same data will include their own description of these data.

Finding the underlying data structures that are described by a collection of partial schemas must take into account the following facts :

- schema redundancy or overlapping : a data structure D can be described in several programs through several views, say S1, S2, .., Sn;
- syntactic variation : the descriptions of D in S1, S2, .., Sn can be different (name, value set, length, structure, etc);

- semantic variation : a data structure D can be perceived with different viewpoints in different programs (a program processes the employees while another one processes the subset of the female employees only);
- complementarity : some data are described in some views but not in others.

However, this situation can be found with more advanced DMS as well. Though these DMS offer a global schema for each of the database they manage, the database related to the application domain can be made up of several operational, physically distinct, databases. Situations in which an IMS database and a relational database coexist are not uncommon. If these databases cover parts of the same application domain, reverse engineering each of them leads to two complementary and overlapping schemas that presents all the characteristics mentioned above.

Solving the problems induced by the existence of multiple views is called schema (or view) integration and schema redundancy reduction.

This point will be developed further in chapter 10.

5.3.4 Schema transformation

Many operations that have to be carried out on data structure descriptions are basically schema transformations, also called schema restructuration. Schema transformation is a basic tool for both database forward and reverse engineering. Indeed, it is the operator through which the designer will

- normalize a conceptual schema,
- restructure a logical schema in order to optimize it according to various criteria,
- translate a logical schema in order to make it comply with a specific DMS.

In addition, schema transformation will allow the reverse engineer to

- get rid of optimization constructs,
- untranslate a DMS-compliant schema,
- restructure a logical or conceptual schema in order to make it more readable.

This point will be developed further in chapter 7.

5.4 THE SOURCES OF INFORMATION

This presentation is mainly oriented towards the exploitation of the information that can be extracted from the operational (i.e. executable) source texts of database description and of application programs. In practice, any source of information is welcome provided it

can provide direct or indirect knowledge on the application domain. Among the most important sources we can mention,

- data entry forms and output report structure, that can be considered as a sort of user's view on the database,
- existing documentation, when it still exists,
- data dictionaries, in which a centralized database description can be found, even for standard file systems,
- CASE tools, the repository of which contains information covering documentation and data dictionary objectives,
- data file contents, that can give hints on structural constraints such as identifiers and referential integrity constraints,
- interviewing the actual developer, when still available, can be one of the richer and more reliable source,
- user interview, either to enrich a draft schema, or to validate a proposed final schema.

This point will be developed further in chapter 14.

5.5 PHYSICAL/CONCEPTUAL MAPPINGS

In many cases, reverse engineering an existing database is a long and complex process in which many data structure transformations will occur, in such a way that the correspondance between the source descriptions and the final schema is far from obvious. A physical object may generate several conceptual objects, while in other cases, several physical objects may be described by one conceptual object only. In addition, a single conceptual object can be physically described in many source texts.

Memorizing and maintaining this correspondance are essential from two points of view.

- First, it can give the conceptual description of any physical data structure, i.e. it can supply the semantics or meaning of actual data.
- Second, it can give information on where and how a conceptual data structure is implemented.

These mapping informations are the basis of the data administration functions. In particular, they ease considerably data conversion, migration and integration.

In realistic situations, the problem of recording this correspondance is not always simple. Besides its many-to-many aspects mentioned earlier, the path between the source data structures and the final data structures can be long. Indeed, a single physical object can have been renamed, transformed several times, integrated with several other objects before reaching its final conceptual state. Therefore, the mapping between both descriptions can be made up of the composition of several elementary mappings.

This point will be developed further in chapter 13.

5.6 LIMITS OF THE METHOD

In many of its aspects, database forward engineering is basically a fairly well formalized process that uses a sort of waterfall model, where the result of each step is the input for the following step. The analysis carried out in section 5.1 is based on the implicit hypothesis that reverse engineering is made up, for most of its activities, by simply reversing the database design processes. This has lead to the general model proposed in 5.2.

Due to the nature of the information available, to the complexity and the so many degrees of freedom that characterize it, reverse engineering a database may not fit so nicely into such a structured framework. Practical experiments have led to the conclusion that when reverse engineering a database, all the activities may be conducted at any time. For instance, going back to the source texts when working on an nearly completed conceptual schema is not uncommon. More flexible strategies than the mere bottom-up one suggested in 5.2 can be needed in difficult situations. It seems that they can be derived from the proposals defined in this chapter.

However, the over-disciplined strategy proposed hereabove provides a reference framework that allows a clear description and discussion of the problems, of their origin and of the elementary activities that can solve them, wherever and whenever they will be carried out. It allows also to analyse specific problems that are important, whatever the strategy. Studying the global process, i.e. proposing actual strategies, cannot be done independently of the source DMS[5]. Therefore, the problem of reverse engineering strategies will be examined in each DMS-dependent part.

[5] this conclusion is obvious when distinguishing, for example, DMS that offer a unique physical schema from those that use partial user's views only.