



Institutional Repository - Research Portal

Dépôt Institutionnel - Portail de la Recherche

researchportal.unamur.be

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Computer-Aided Database Engineering

Hainaut, Jean-Luc

Publication date:
2002

[Link to publication](#)

Citation for published version (HARVARD):

Hainaut, J-L 2002, Computer-Aided Database Engineering: Database Models..

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

DB-MAIN Manual Series

Computer-Aided Database Engineering

Volume 1: Database Models

Fourth Edition - 1999

The University of Namur - Institut d'Informatique

Credits

To be written

Contacts

Professor Jean-Luc Hainaut
University of Namur - Institut d'Informatique
rue Grandgagnage, 21 λ B-5000 Namur (Belgium)
jlhainaut@info.fundp.ac.be - <http://www.info.fundp.ac.be/~dbm>

INTRODUCTION

to be written

Volume 1: Database Models

Table of contents

1. Building our first database

1.1 Introduction	2
1.2 Starting DB-MAIN	3
1.3 Opening a new project	3
1.4 Defining a new schema	5
1.5 Defining entity types COMPANY and PRODUCT	7
1.6 Entering entity type attributes	8
1.7 Entering relationship type MANUFACTURES	10
1.8 Defining entity type identifiers	12
1.9 Documenting the schema	13
1.10 Producing a SQL database	14
1.11 Saving the project	16
1.12 Quitting DB-MAIN	16

2. A closer look at schemas

2.1 Starting Lesson 2	2
2.2 On including database schemas into a document	2
2.3 Graphical views of a schema	3
2.4 Textual views of a schema	6
2.5 Application: far jumps through a graphical schema	10

3. An even closer look at schemas

3.1 Starting Lesson 3	2
3.2 Securing our work	2
3.3 Manipulating the graphical components of a schema	3
3.3.1 Moving objects	3
3.3.2 Aligning objects	4
3.3.3 Zooming in and out	6
3.3.4 Fonts	7
3.3.5 Grids	7

3.3.6 The Reduce function	7
3.3.7 Colors	7
3.3.8 Marking objects	8
3.3.9 Auto-draw	9
3.3.10 Using a larger schema	10
3.3.11 Last observations	10
3.4 Navigation through textual views	10
3.5 Reordering attributes and roles	12
3.6 Generating reports	12
3.7 Copying objects	15
3.8 Inspecting objects	16
3.9 External links	17
3.10 Quitting the lesson	18
4. Multi-product projects	
4.1 Starting Lesson 4	2
4.2 Conceptual and logical schemas	2
4.3 SQL code generation	6
4.4 Generating reports	8
4.5 Multi-product project	8
4.6 Deleting objects	10
4.7 Export/import of schema components	10
4.8 Why to export schemas?	12
5. The basics of conceptual modeling	
5.1 Starting Lesson 5	2
5.2 Updating an object	2
5.3 What is a conceptual schema?	2
5.4 Cardinality of an attribute	3
5.5 Atomic and compound attributes	5
5.6 Multiple identifiers	7
5.7 Hybrid identifiers	8
5.8 On defining identifiers	10
5.9 N-ary relationship types	11
5.10 Relationship types with attributes	12

5.11 Relationship types with identifier(s)	12
5.12 Cyclic relationship types	15
5.13 The complete schema	18
5.14 On the cardinalities of rel-types	20
5.14.1 Binary rel-types	20
5.14.2 N-ary rel-types	22
5.15 Minimal identifiers	22
5.16 What next?	23
6. The basics of logical and physical modeling	
6.1 Introduction	2
6.2 What is a logical schema?	2
6.3 Transformation into a logical schema	3
6.4 Reference attributes (foreign keys)	6
6.5 Access keys	11
6.6 On the conceptual → relational translation rules	13
6.7 Defining entity collections	17
6.8 Name processing	18
6.9 SQL code generation	22
6.9.1 About the coding rules	25
6.9.2 On SQL generation styles	26
6.9.3 The Voyager 2 meta-development environment	26
6.10 About the DB-MAIN graphical representation	27
6.11 Logical vs physical schemas	28
6.12 Closing the lesson	29
7. Names	
7.1 Introduction	2
7.2 Uniqueness rules	2
7.3 Ambiguous names (the symbol)	3
7.4 How to choose names	4
7.5 Name processing	8
7.6 Changing the prefix of names	11
7.7 Lexicons	11

8. More about entity types

8.1 Starting Lesson 8	2
8.2 Classification hierarchies (IS-A relations)	2
8.3 Properties of the subtypes of an Entity type	5
8.4 Supertype/Subtype inheritance	8
8.5 Multilevel IS-A hierarchy	11
8.6 Multiple inheritance	13
8.7 Processing units of a schema	18
8.8 Quitting DB-MAIN	19

9. More about attributes

9.1 Introduction	2
9.2 Built-in domains	2
9.3 User-defined domains	4
9.4 Stable and non-recyclable attributes	7
9.5 Attribute identifiers	9
9.6 Non-set multivalued attributes	12
9.6.1 Sets	14
9.6.2 Bags	14
9.6.3 Unique lists	15
9.6.4 Lists	15
9.6.5 Arrays	16
9.6.6 Unique arrays	17
9.6.7 Summary	17
9.6.8 Set expression of non-set multivalued attributes	18
9.7 Multivalued identifiers	21
9.8 More on access keys	23
9.9 Multivalued reference attributes	24
9.10 Non-standard reference attributes	26
9.10.1 Hierarchical foreign key to a multivalued attribute	27
9.10.2 Overlapping foreign keys	27
9.11 Object attributes	28

10. More about constraints

10.1 Introduction	2
-------------------	---

10.2 Existence constraints	2
10.3 Coexistent components of an entity type	2
10.4 Exclusive components of an entity type	5
10.5 Groups with at least one, or exactly one, existing component	8
10.6 Existence constraints rules	9
10.7 Existence constraints and IS-A relations	12
10.8 Other existence constraints	15
10.9 Generic constraints	16
10.9.1 Generic group constraints	16
10.9.2 Generic inter-group constraints	18
10.10 Schema transformation: another look	19
11. More about relationship types	
11.1 Introduction	2
11.2 Multi-ET roles	2
11.3 Generic rel-types	4
11.3.1 Aggregation	5
11.3.2 Topological relationships	7
12. View schemas	
12.1 Introduction	2
12.1.1 When to use views?	2
12.1.2 Principles	3
12.2 Specifying the objects of the view	3
12.3 Defining the view	5
12.4 Displaying a latent view	5
12.5 Materializing a view as a view schema	6
12.6 Modifying a view schema	8
12.7 What if I change my mind about the view?	9
12.8 Modifying the source schema	10
12.9 Propagating the modification of the source schema to view schemas	11
12.10 Warning	12
12.11 Other operations	14
12.12 Technical information	14
12.13 The View Menu	15

12.14 There are views and views!	15
----------------------------------	----

13. Text Processing

13.1 Introduction	2
13.2 Text file manipulation	2
13.2.1 Selecting and marking text lines?	3
13.2.2 Line annotation	6
13.2.3 Reports from text files	7
13.3 Text structure and text analysis	8
13.4 Natural language analysis	8
13.5 DDL physical schema extraction	8
13.6 Patterns	10
13.7 Dependency graphs	11
13.8 Program slice	12

14. Conceptual Models (*to be completed*)

14.1 Introduction
14.2 Entity-relationship models
14.3 Object-role models
14.4 Object-oriented models
14.5 The UML class model (conceptual)

15. Logical Models (*to be completed*)

15.1 Introduction
15.2 Relational models
15.3 Object-oriented models
15.4 The UML class model (logical/physical)
15.5 Object-relational models
15.6 CODASYL DBTG models
15.7 Hierarchical models
15.8 Shallow models
15.9 Standard file models

16. Miscellaneous

16.1 Introduction	2
16.2 Generic properties	2
16.2.1 Semi-formal properties	3
16.2.2 Dynamic properties	5
16.3 Configuration settings	9

Appendix A: The Generic DB-MAIN Model

A.1 The specification model in short
A.2 Project
A.3 Base schema
A.4 View schema
A.5 Text file
A.6 Inter-product relationship
A.7 Entity type (or object class)
A.8 Relationship type (rel-type)
A.9 Collection
A.10 Attribute
A.11 Object-attribute
A.12 Non-set multivalued attribute
A.13 Group
A.16 Inter-group constraint
A.15 Processing units
A.16 Common characteristics
A.17 Names
A.18 Structure of a text file

Lesson 1

Building our first database

Objective

In this first lesson, the reader will learn how to start and quit the DB-MAIN CASE tool, how to introduce a simple Entity-Relationship conceptual schema, and how to translate it into table and column structures expressed into the SQL language. S/he will also save her/his work for further use.

Above all, the reader will get an insight into what *Database Design* is all about.

Preliminary checking

For this lesson, be sure that the DB_MAIN directory includes the DB_MAIN.EXE program (the CASE tool) as well as all the run-time libraries (*.dll). See the README.TXT file for further detail.

This lesson assumes that you use DB-MAIN Version 5, but is valid for version 4 as well.

1.1 Introduction

We will develop a very simple database intended to describe companies that manufacture products. Through this process we will familiarize ourselves with some important concepts in database engineering.

For instance, we will learn that besides the data structures that are built in the computer, and in which we will store the data about these *companies which manufacture these products*, there exists another, more abstract and more intuitive way to describe these concepts, namely the **conceptual schema**. While data are stored into tables or into files, a conceptual schema describes the concepts in terms of entity types (classes of similar objects), attributes (entity properties) and relationship types (associations holding among entities).

The most straightforward **conceptual schema** comprises the entity type COMPANY, which describes the class of companies, and the entity type PRODUCT, representing the class of products. The fact that companies manufacture products is represented by a *many-to-one* relationship type called *manufactures* connecting their entity types. We will give these entity types some attributes that describe the properties of the companies (such as their company identifier, their name and their revenue) and of the products.

1.2 Starting DB-MAIN

Through the Explorer (or File Manager), we go into the DB_MAIN directory, and we start the DB_MAIN program by double-clicking on the DB_MAIN.EXE name or on the DB-MAIN icon. We acknowledge the presentation box by clicking on the OK button, or by pressing the Enter key. The main DB-MAIN window appears, showing, among others, the *Menu bar* (with two items only: **File** and **Help**), the *Tool bar* (with a few buttons, among which

are build a new project and open an existing project), the *Workspace*, in which the project window will be displayed (currently empty), and the *Status bar*.

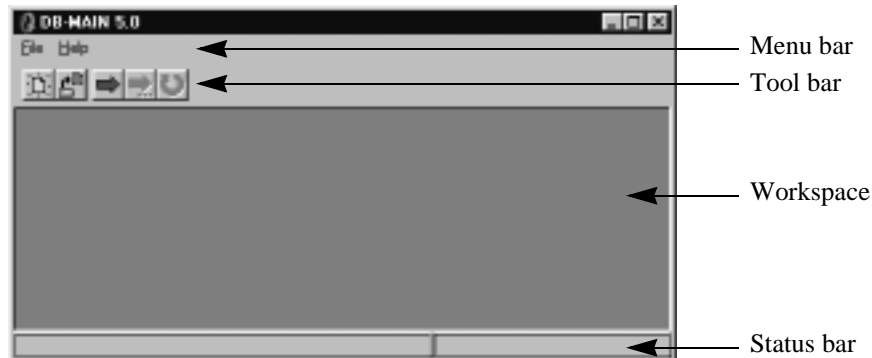


Figure 1.1 - The main window of DB-MAIN.

1.3 Creating a new project

We are ready to open a new project through the command **File / New project**. This command opens a *Project Property box* (or *Project box* for short), which asks us some information about the new project. Our project will be called MANU-1 and will be given the short name M1. We validate the operation by clicking on button OK.

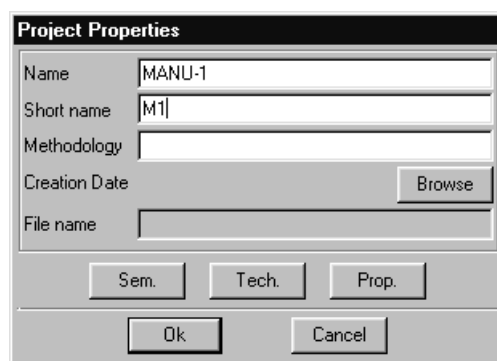



Figure 1.2 - The properties of the new project.

Note 1. There is a simpler way to open a new project, namely by pressing the New project button  in the Tool bar.

Now, a new window, namely the *Project window*, appears in the DB-MAIN workspace. Currently, it includes a small rectangle, which is the iconic representation of the project itself (any DB-MAIN object has a graphical representation). To examine its properties, try **File / Project Properties**¹. Later on, this window will also show all the products of the project, such as the various schemas and texts, together with their relationships².

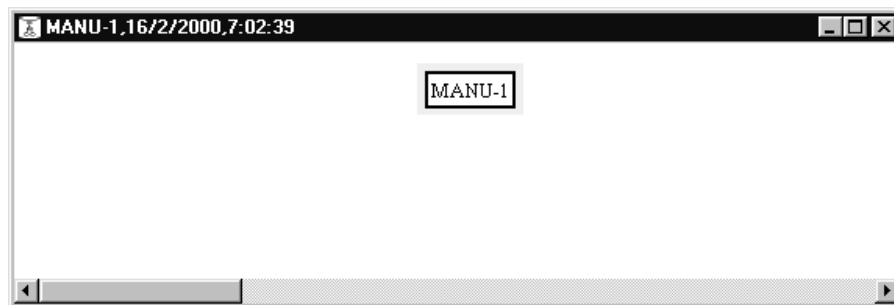


Figure 1.3 - The project window in which all the documents of the project will appear.

The Menu bar and the Tool bar have changed too, offering more functions that will be used later on. Make sure that the *Standard tools bar* is available. Otherwise use **Windows / Standard tools** to make it visible.



Figure 1.4 - The complete Menu bar and the full *Tool bar*.

-
1. Double-clicking does not work here, for reasons that will be explained later.
 2. This window can also show all the activities that have been carried out to build these products. In other words, the Project window can show, if requested to, the history of the project. We will ignore this feature in the following lessons.

1.4 Defining a new schema

We create a new schema in which we will draw the conceptual structures of the database. Through the command **Product / New schema** the *Schema box* appears and asks us the name (Manufacturing), the short name (Manu) and the version of the schema. This schema will include the conceptual description of our database in project, so that `Conceptual` should be a clear version name that suggests the objective of the schema.

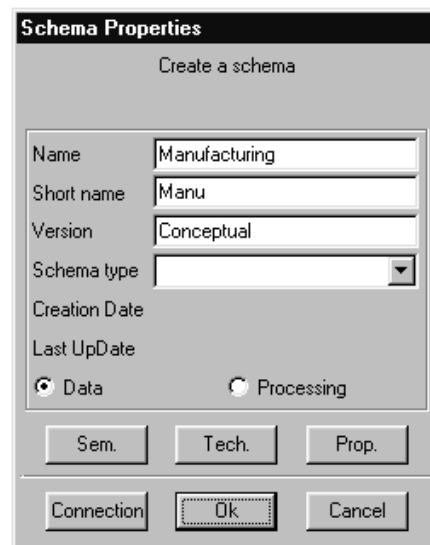


Figure 1.5 - Creating a new schema.

We ignore the other properties and we validate the operation by clicking on the OK button.

Two things happen. First, a new icon with the name `Manufacturing/Conceptual` appears in the Project window, indicating that the project comprises a new document, or product, which is a schema. Later on, double-clicking on such an icon will open its *Schema window*.

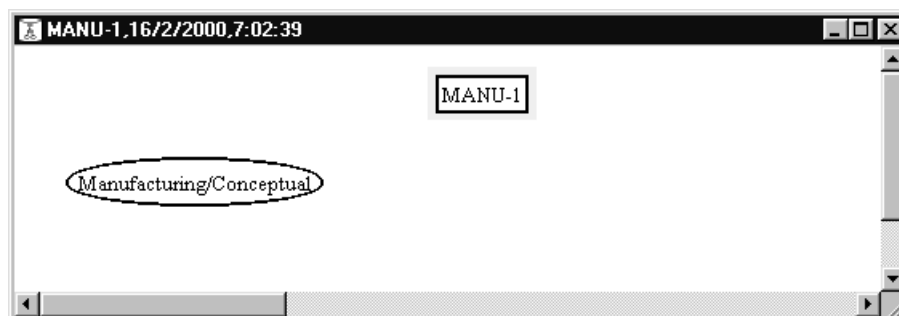


Figure 1.6 - The project window includes the new schema³.

Secondly, a *Schema window* is opened, showing the same icon, but nothing else.

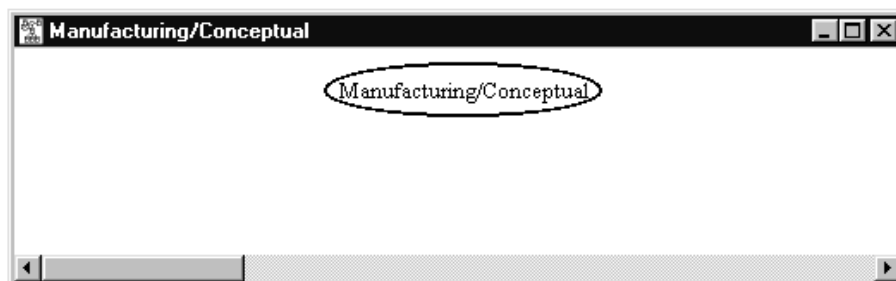


Figure 1.7 - The schema window is empty, except for the icon of the schema itself. This window is like a blank page on which we will draw the conceptual schema of the future database.


This icon represents the schema. Double-clicking on it opens its *Schema (property) box*. So far, this schema is empty. We will work in this window, so that it is a good idea to enlarge it.

-
3. In some rare situations (for instance, if you work on a DB-MAIN version already used by a professional who configured it differently) a small rectangle with the label New schema also appears in the Project windows. To get rid of it, check that the Project window display mode is *Graphical Dependency* (through **View / Graph. Dependency**). The other modes are quite nice as well, but probably a bit disturbing for an introductory lesson!

From now on, in order to simplify the illustrations used in this lesson, we will hide the schema object, except when needed.

Note. To free the workspace, especially when it is crammed with many windows, it is best to iconize (minimize) the Project window.

1.5 Defining entity types COMPANY and PRODUCT

To enter the *create entity type* mode, we click on the  button. That changes the cursor that now looks like a little rectangular box. We choose a point in the schema window, we put the cursor on it and we double-click. This lays an entity type at that point and opens the *Entity type box* that allows us to define a new entity type (Figure 1.8).

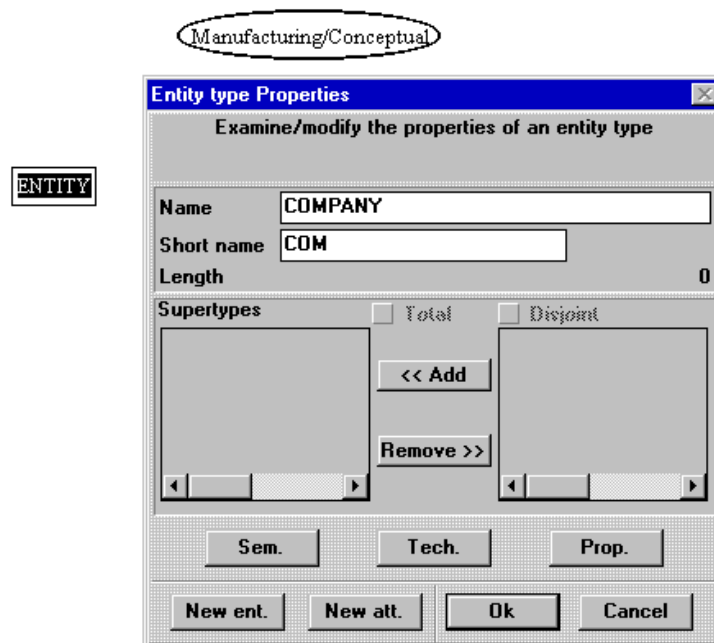


Figure 1.8 - The first entity type is defined.

We enter the name `COMPANY` and short name `COM`. We validate the operation by clicking on the `OK` button.

In the same way, we double-click at another point to define entity type `PRODUCT` with short name `PRO`. To quit the entry mode, we click on the *New Entity type* button again, or we press the `Escape` key.

Now, the schema window shows the newly defined entity types as two boxes. We move the boxes (by dragging them with the mouse) in the window in order to give the schema a nice layout (Figure 1.9)

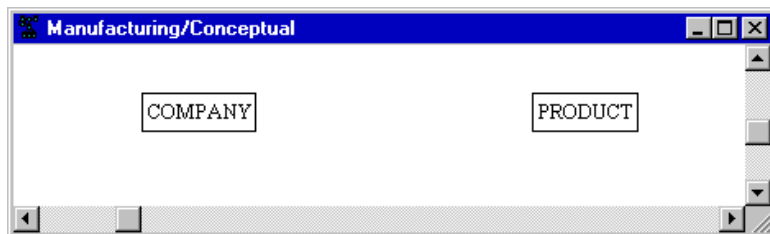


Figure 1.9 - So far, the current schema is made up of two entity types.

1.6 Entering entity type attributes

To specify that some specific information items are associated with the entities of each type, we will define the *attributes* of these entity types. We open the property box of entity type `COMPANY` by double-clicking on its name in the schema window, then we click on the `New att.` button. The *Attribute box* invites us to define the first attribute (Figure 1.10). We give it the name `Com-ID`, the type `char(acter)` and the length 15. This attribute represents the company identifier, and is considered as a string of 15 characters. For now, we can ignore the other properties.

There are other attributes that we want to associate with `COMPANY`. Therefore, we click on button `Next att(ribute)`, which validates the current definition, and which calls the *Attribute box* again (since this button is the active one, just pressing the `Enter` key will do it). We define successively attributes `Com-Name` (`char 25`), `Com-Address` (`char 50`) and `Com-Revenue` (`numeric 12`). The last attribute will be validated by clicking on the `Ok` button instead to stop the entry process.

The dialog box is titled "Attribute Properties" and has a subtitle "Create attribute of COMPANY". It contains the following fields and controls:

- Name: Text box containing "Com-ID"
- Short name: Empty text box
- Cardinality: Dropdown menu set to "1-1"
- Type: Dropdown menu set to "Char"
- Stable: Unchecked checkbox
- Non Recyclable: Unchecked checkbox
- Length: Spin box set to "15"
- Buttons: "Sem.", "Tech.", "Prop.", "First att.", "Next att.", "Ok", "Cancel"

Figure 1.10 - The first attribute of COMPANY is defined. The next attributes will be defined by pressing the `Next att.` button, or more simply by pressing the `Enter` key.

In the same way, we define attributes `Pro-ID` (char 8) and `Pro-Name` (char 25) of entity type `PRODUCT`.

The schema window now looks like Figure 1.11.

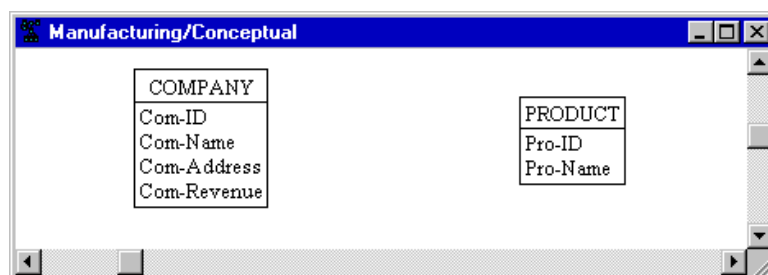



Figure 1.11 - The entity types have been given specific attributes.

1.7 Entering relationship type MANUFACTURES

Now we want to represent the fact that *companies manufacture products*. This can be done by drawing a relationship type (or *rel-type* for short) between these entity types.

We enter the *New rel-type* mode by clicking on the button  in the Tool bar⁴. The cursor takes a cross-hair shape, so that we can draw a line from COMPANY to PRODUCT in the schema window (Figure 1.12).

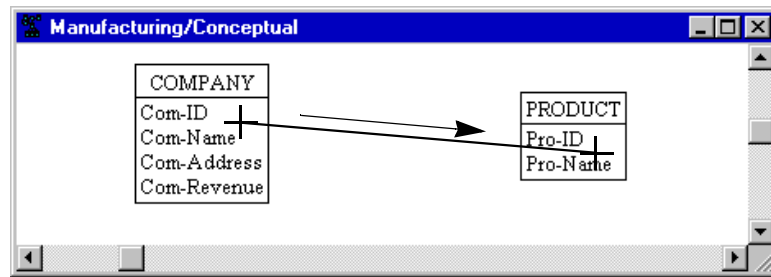



Figure 1.12 - A line is drawn between the boxes of the entity type we want to connect.

A link appears between both rectangles with a hexagon on it. Normally, the default name R is selected (white on black). If it is not, we click on it. We press the Enter key to open the *Rel-type box* (or we double-click on name R). We enter the correct name *manufactures*, then we validate through the Ok button (Figure 1.13).

We quit the entry mode just like we did for the entity types by pressing the Escape key or by clicking on the button  again (or on any another entry button).

Each end of the rel-type is called a **role**. Each role is taken by an entity type and is given a *cardinality constraint*, that appears as a pair of symbols, such as 0-N and 1-1.

The 0-N cardinality specifies that any COMPANY entity will appear in at least 0 and at most N (standing for *infinity*) manufactures relationships.

4. or button  in Version 3.

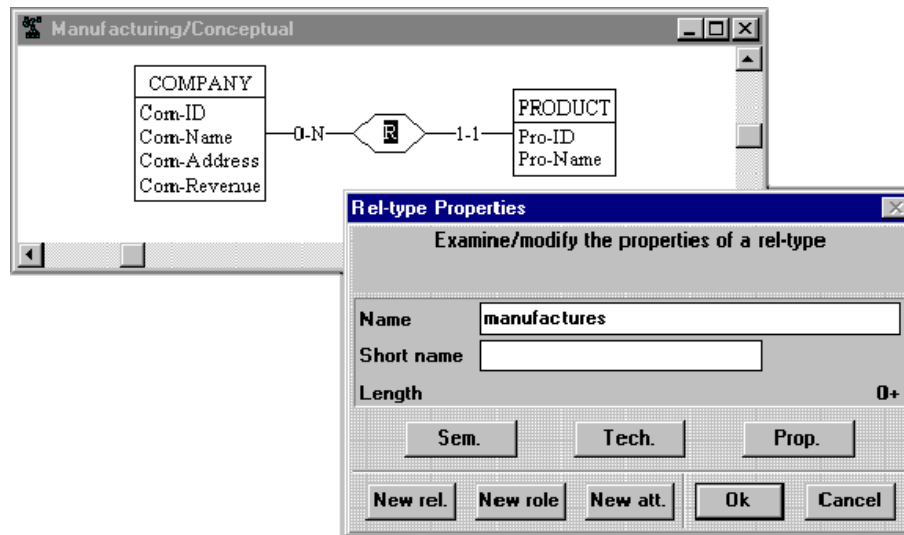


Figure 1.13 - A relationship type links the entity types. It will be given the name *manufactures*.

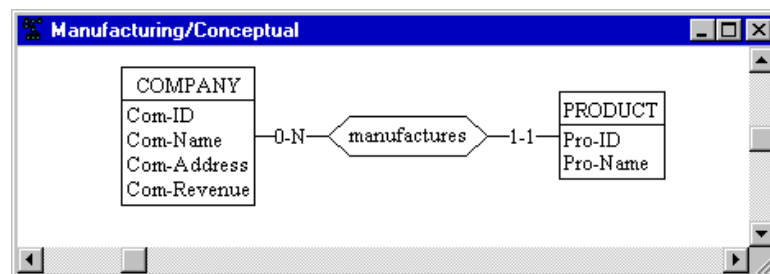



Figure 1.14 - Now the schema explicitly tells that *companies manufacture products*.

We will study later the concept of cardinality in greater detail. For now, we understand the 0-N cardinality as "*a company manufactures an arbitrary number (i.e., from 0 to N) of products*". Similarly, the schema shows that a PRODUCT entity will appear in exactly one (i.e., from 1 to 1) manufactures relationship.

The cardinality can be changed by double-clicking on the role, i.e., on its cardinality symbol. This will be examined in detail in another lesson.

1.8 Defining entity type identifiers

Normally, the entities of the same class, for instance all the companies, have a special property that allows us to designate each of them. This property is called an **identifier** of the entity type. Usually, it is a name, a code, a reference or anything else that makes the entities unique in their class.

For instance, we want to tell that Com-ID is the unique code of companies. We select this attribute by clicking on its name (which appears white on black) than we click on the Identifier button  on the Tool bar.

In the same way, we define PRO-ID as the identifier of entity type PRODUCT. The schema can now be considered as complete (Figure 1.15).

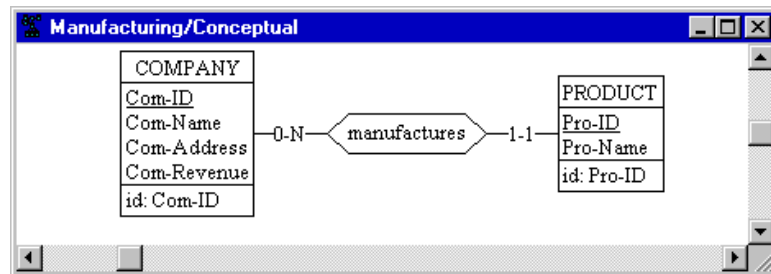


Figure 1.15 - An identifier has been associated with each entity type.

Note that the identifier is graphically mentioned twice (assuming the novice analyst has not noticed the fact!): first through the `id` clause that appears at the bottom of the entity type box, and secondly by the underlining of the component attribute. This latter way will be used when the identifier comprises attributes only.

1.9 Documenting the schema

You have probably observed that most boxes that define the properties of an object have a special button named Sem. Clicking on the Sem button opens a

small text window in which we are allowed to enter a free text that describes the meaning of the current object, i.e., its *semantics*.

Let us double-click on the COMPANY entity type (another way: select COMPANY, then press the Enter key). We get the Entity type property box of COMPANY. We click on the Sem button, and we enter a text that defines what a company is (Figure 1.16).

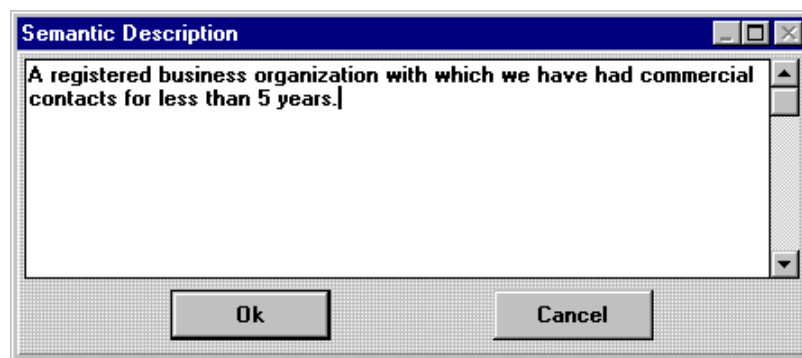



Figure 1.16 - The *Semantic description* text window of an object.

The text can be as long as needed (with a 64 Kb limit however). It can be cut, copied and pasted from/to any other program in the usual way (ctrl-X, ctrl-C, ctrl-V).

In the same way, we can enter a description for PRODUCT and manufactures, for each of the attributes, for each role, for each identifier and even for the schema and the project themselves.

Note. There is a similar button  on the *Standard tools* bar which has the same effect: select any object in the current schema, then click on this button to open the Semantic description window of the object. ✓

1.10 Producing a SQL⁵ database

There are several ways in which this conceptual schema can be translated into table and column structures. For now, we have no special requirements as far as performance, or any other consideration, are concerned. We will be happy with an unsophisticated translation of this schema into SQL commands.

This translation can be done in a straightforward way through the command **Transform / Quick SQL**. DB-MAIN simply asks you, with the standard file dialog box, in which file you want the SQL program to be stored. By default, the file will be named `manu-1.ddl`, following the name of the project (Figure 1.17).

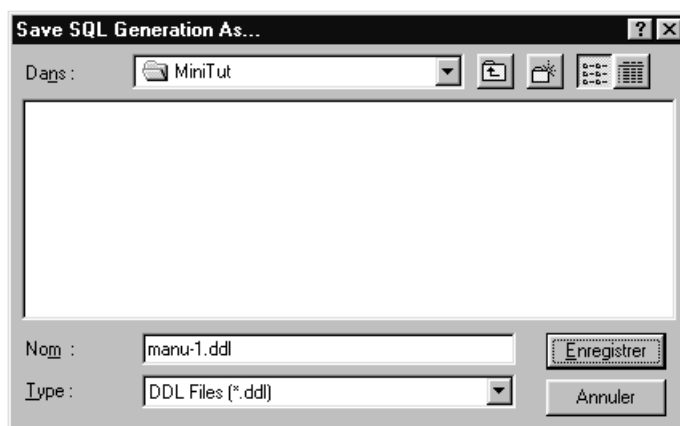


Figure 1.17 - The SQL program that is being generated from the conceptual schema will be saved as `manu-1.DDL` file.

Now, we go back to the Project window. We observe that a new product has been made available. The slightly different icon shape indicates that this new document is a text file called `manu-1.ddl`. Obviously, this is the SQL program we just generated in the last step.

We can examine the contents of this text file by double-clicking on its icon. A new text window opens, showing the SQL code implementing the conceptual schema. It should read like in Figure 1.19.

5. SQL must be read SEQUEL.

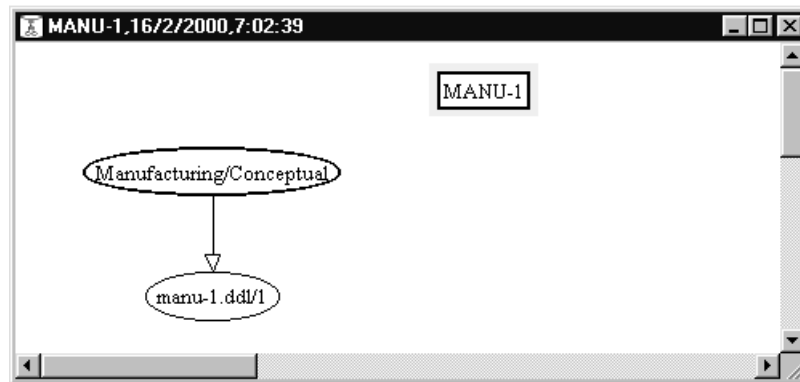


Figure 1.18 - Now, the project window includes two products, namely the conceptual schema and the SQL program that derives from it.

```

create database Manufacturing;

create table COMPANY (
  Com-ID char(15) not null,
  Com-Name char(25) not null,
  Com-Address char(50) not null,
  Com-Revenue numeric(12) not null,
  primary key (Com-ID));

create table PRODUCT (
  Pro-ID char(8) not null,
  Pro-Name char(25) not null,
  Com-ID char(15) not null,
  primary key (Pro-ID));

alter table PRODUCT add constraint FKmanufactures
  foreign key (Com-ID) references COMPANY;

create unique index IDCOMPANY on COMPANY (Com-ID);
create unique index IDPRODUCT on PRODUCT (Pro-ID);
create index FKmanufactures on PRODUCT (Com-ID);

```

Figure 1.19 - The contents of the manu-1.ddl text file can be examined by double-clicking on its icon in the project window.

To be quite precise, this SQL program will not necessarily be executable on all machines, and would probably need some syntactic adjustments. For instance, dashes ("-") are not allowed by most SQL DBMS, and should be replaced by, say, underscores ("_"). We will see later how this kind of problem can be addressed in a systematic way.

In addition, the set of indexes may not be the most efficient one, and would need some refinement. Such decisions relate to physical design, an activity that obviously is far beyond the scope of this first lesson!

1.11 Saving the project



As is natural after working such a long time, we carefully save our work through command **File / Save project** (or  button) or command **File / Save project as** (or  button) in order to make it available for further use.



Figure 1.20 - The whole project is saved on disk.

By default, the project is saved as file manu-1.lun. We validate the operation through the button OK.

The *.lun extension is typical to the saved DB-MAIN projects, so do not use them for other files.

1.12 Quitting DB-MAIN

It is now time to exit from the DB-MAIN tool by command **File / Exit**.








We have built our first SQL database, and we are now able to build other simple SQL databases just by applying the basics that have been presented in this lesson.

Key ideas of Lesson 1

1. A **CASE** (Computer-Aided Software Engineering) **tool** is a software that allows a developer to draw the conceptual schema of an application domain, then to generate the SQL tables that represent this application domain.
2. The **application domain** is that part of the real world about which we want to collect, maintain and process information. This information will be represented by data stored in the database of the application domain.
3. A **database** is a collection of data that codes facts about the application domain. At the present time, most databases are organized into relational tables. A table is made up of columns; some of which can be declared its primary key. Indexes can be associated with each table.
4. A **conceptual schema** is the computer-independent description of the facts that make up an application domain. It comprises entity types, attributes, relationship types (or rel-types) and identifiers. An entity type describes a class of significant concrete or abstract objects of the application domain. An attribute represents a property common to the entities of a given type. A rel-type represents a class of associations between entities.
5. The CASE tools can turn a conceptual schema into a database schema. It stores both schemas so that they can be used again later on.

Summary of Lesson 1

- In this first lesson, we have studied some important concepts:
 - the concept of CASE tools
 - projects and schemas
 - entity types, relationship types, attributes and identifiers
 - conceptual schemas
 - SQL expression of a conceptual schema

- We have also learned to:
 - run the DB-MAIN CASE tool
 - create a new project: **File / New project** 
 - create a new schema: **Product / New schema**
 - define an entity type: **New / Entity type** 
 - define an attribute: **New / Attribute**
 - define a relationship type: **New / Rel-type** 
 - define an identifier: **New / Group** 
 - add a semantic description: 
 - save the current project: **File / Save as** 
 - save the current project: **File / Save** 
 - produce SQL code: **Quick DB / SQL**
or **Transform / Quick SQL**
 - exit from DB-MAIN: **File / Exit**

- We have produced two types of files:
 - saved projects (* . lun)
 - executable code such as SQL (* . ddl).

Exercises for Lesson 1

Define a project, a conceptual schema and generate an SQL database creation program for each of the situations described below.

- 1.1 The small database we developed in this lesson was based on the hypothesis that a product is manufactured by one company only (cardinality 1-1). Now, consider that *a product can be produced by any number of companies* (i.e., by 0, 1, 2, or more companies). Change the schema accordingly. Don't save this project.
- 1.2 Customers buy products in such a way that each customer can buy any number of products and each product can be bought by an arbitrary number of customers. Imagine some natural attributes for the entity types. Call this project SALES1 and save it.
- 1.3 Students belong to classes: each student belongs to exactly one class (no less, no more), while a class comprises any number of students. Each student can be registered in any number of courses while any number of students can be registered for a given course. Imagine some natural attributes for the entity types. Call this project STUDENT1 and save it.
- 1.4 Complete the MANU-1 project by considering countries to which products are exported.
Don't save the modified project (we will make use of the original version in further lessons), unless you give it another name.

Lesson 2

A closer look at schemas


Objective

This is an easy and relaxing lesson (just playing with existing schemas!). It presents some useful schema display formats and the way to use them.

Preliminary checking

In this lesson, we will use the project MANU-1 (file manu-1.lun) that has been created in Lesson 1, and the LIBRARY project (or its French equivalent BIBLIO) that comes with the DB-MAIN software.

2.1 Starting Lesson 2

Let us start DB-MAIN and open the MANU-1 project through the command **Project / Open project** or by clicking on the button . When the project is opened, we double-click on the icon of the Manufacturing/Conceptual schema to display its contents.


For this lesson, we will need some new functions that are offered by the menu, but that are available on a new tool palette as well. We display this new palette through **Windows / Graphical tools** (Figure 2.1). These tools can be placed anywhere on the screen, for instance under the *Standard tool bar*.



Figure 2.1 - The graphical tool bar. It can be resized according to your taste.

2.2 On including database schemas into a document

In the first lesson, several figures include a schema, showing the step-by-step construction of the conceptual description of our database. As everybody should have observed, these schemas have been obtained from screen copies. This technique provides nice looking results, but is rather painful (the screen shots have to be processed with an image processing software) and yields huge documents.

The DB-MAIN tool includes a function that copies selected schema objects onto the clipboard in a more concise format (as vector-based objects). So, select all the objects of the schema, then call the **Edit / Copy graphic** menu item or click on the  button in the *Graphical tools* bar. Then, open a Word or

Powerpoint document, and paste the clipboard contents (use *Paste* or *Paste Special* according to the software).

The schema objects appear in the text document as in Figure 2.2 (bottom). The result can be modified as any vector-based graphical object¹. From now on, we will use this technique to include schema fragments in this lesson and in the next ones.

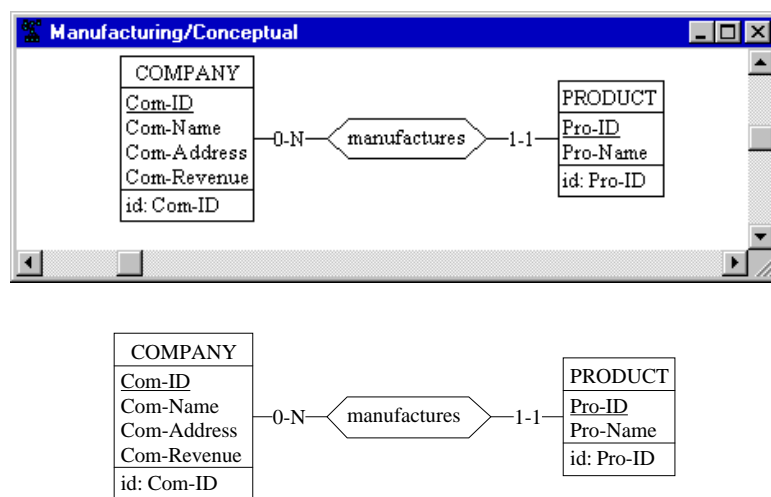


Figure 2.2 - Bitmap (top) and vector-based (bottom) schemas as they appear in a text document.

2.3 Graphical views of a schema

In Lesson 1, the schema was represented in a Schema window through graphical objects. There are several other ways to display this schema. They can be classified into *graphical views* and *textual views*. This section is devoted to graphical views.

1. In some products, such as MS-Word or FrameMaker, the labels may appear to be too long or too short for the rectangles in which they are enclosed after the schema has been redimensioned. This is due to the way Windows redimensions a graphical object: continuously for geometrical components and point by point for texts. In this case, just expand or stretch the schema frame **horizontally** until the texts correctly fit in their boxes.

Let us first examine a new way of presenting large schemas, namely the *compact view*. It can be obtained through the **View / Graph. compact** command. The attributes and identifiers are hidden in such a way that only the schema *skeleton* appears (Figure 2.3).

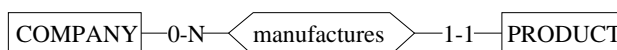



Figure 2.3 - The *compact graphical* view of the MANU-1/Conceptual schema.

Now, we go back to the standard graphical view through **View / Graph. standard**, to get the view we have used so far (Figure 2.4). Since this view is the most useful, it has been given a special button on the Standard tools bar: .

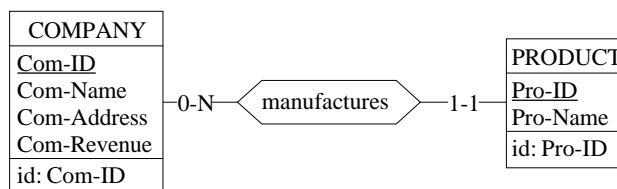


Figure 2.4 - The *standard graphical* view of the MANU-1/Conceptual schema.

Starting from this standard view, we can derive some simplified forms by using the graphical settings panel (**View / Graphical settings**) (Figure 2.5).

The buttons of the *Show Objects* block of this panel can be unchecked, which hides the attributes, or the identifiers (called *groups* in the panel), or both (Figure 2.6). You can also show the attribute types if needed.

Graphical variants exist to represent entity types and rel-types. For instance, we can choose to draw entity type and/or rel-type boxes with round corners instead of square ones by selected *rounded* shape in the Graphical settings panel (Figure 2.7). These settings are valid for the current schema. They can be useful to distinguish different levels of schemas.

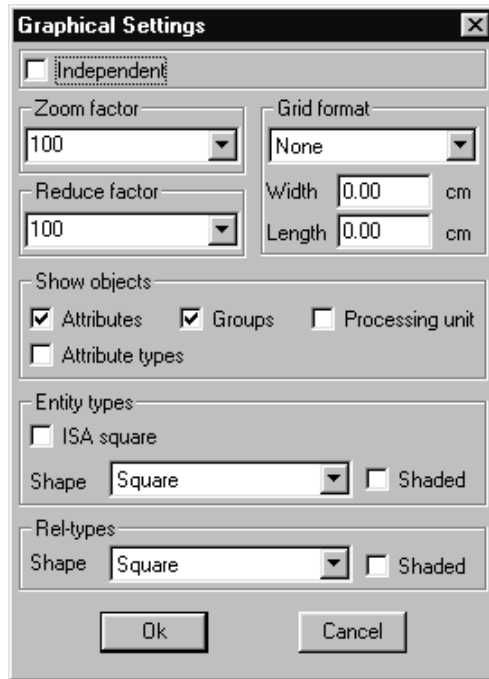


Figure 2.5 - The graphical settings panel.

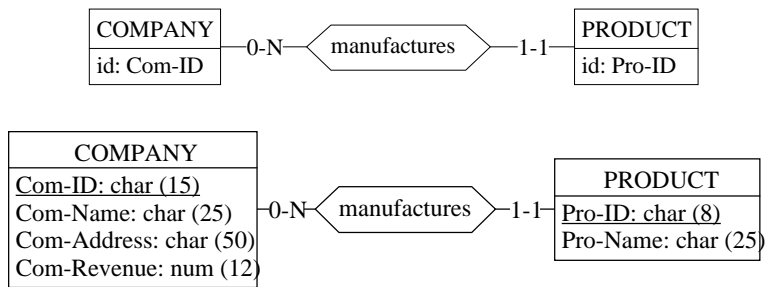


Figure 2.6 - The Standard view without Attributes (top) and without Groups (i.e., without identifiers) but with attribute types (bottom).

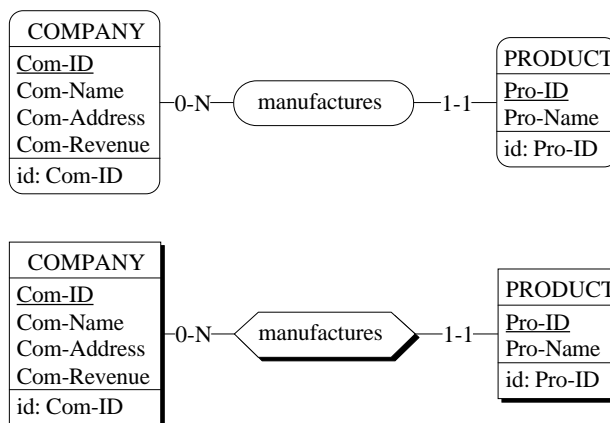


Figure 2.7 - Round-corner shape and shaded boxes as alternate graphical representations.

A last trick before leaving the graphical views of a schema: *how to retrieve a selected object in a schema*. Let us suppose that the (small) schema window shows a fragment of a (large) schema. Let us also suppose that an object is selected, somewhere in the schema, but not shown in the window. How to move the window in such a way that the selected object is at the center of this window? Nothing can be simpler: just press the **tab** key.

What if there is more than one selected object? The **tab** key brings the *next selected object* to the window.

2.4 Textual views of a schema

The contents of a schema can be presented as a pure text as well. In this mode, four formats are available.

The simplest one is the *compact view*. It shows a mere list of the names of the entity types followed by that of the relationship types (Figure 2.8).

```

Schema Manufacturing/Conceptual

COMPANY
PRODUCT


manufactures

```

Figure 2.8 - The **Text compact** view of a schema

This list is a sort of dictionary. It can be obtained through the command **View / Text Compact**.

The compact view does not display the detail of a schema and can be used as a quick index to locate an object in a large schema.

For a more detailed textual view, try the *Standard view*. It can be obtained through the command **View / Text Standard**, and presents the current schema as in Figure 2.9. Since it is frequently used, it can also be obtained through a specific button on the Standard tools bar: .

```

Schema Manufacturing/Conceptual

COMPANY
  Com-ID
  Com-Name
  Com-Address
  Com-Revenue
  id: Com-ID

PRODUCT
  Pro-ID
  Pro-Name
  id: Pro-ID

manufactures (
  [1-1] : PRODUCT
  [0-N] : COMPANY)

```

Figure 2.9 - The **Text standard** view of a schema

The *extended view* is an even more complete presentation. In addition to the information of the standard view, the *extended view* shows, among others, the short names, the type and length of the attributes and the roles in which each entity type appears. The symbol [S] indicates that a semantic description has been associated to the object.

This view is obtained through the command **View / Text extended**, and appears as in Figure 2.10.

```

Schema Manufacturing/Connceptual / Manu [S]

COMPANY / COM [S]
  Com-ID char (15) [S]
  Com-Name char (25) [S]
  Com-Address char (50) [S]
  Com-Revenue numeric (12) [S]
  id: Com-ID
  role: [0-N] in manufactures

PRODUCT / PRO [S]
  Pro-ID char (8) [S]
  Pro-Name char (25) [S]
  id: Pro-ID
  role: [1-1] in manufactures

namufactures [S] (
  [1-1] : PRODUCT
  [0-N] : COMPANY)

```

Figure 2.10 - The **Text extended** view of a schema. The directed arcs show the possible jumps through the hyperlinks activated by a right-button click.

Note that the *role lines* that appear both in the entity type and rel-type paragraphs makes it possible to navigate through the whole schema by jumping from an entity type to the relationship types in which it appears, and conversely:

- to jump from an entity type to one of its relationship types: click on the line of the role in the entity type paragraph *with the **right** button* of the mouse.
- to jump from a relationship type to one of its entity types: click on the line of the role in the rel-type paragraph *with the **right** button* of the mouse.

These *hyperlink* functions are very handy for large schemas. More on schema navigation later on in Lesson 3.

The last format is the *sorted view*, which presents an unstructured sorted list of all the names that appear in the schema, together with their type and origin. This view is particularly important for large and complex schemas, specially in *reverse engineering* activities². It can be used too when checking names in conceptual analysis. In addition, it is the easiest way to retrieve an object when only its name is known.

The sorted view can be obtained through the command **View / Text sorted**, and appears as in Figure 2.11.

Schema Manufacturing/Conceptual	
Com-Address	Att. of COMPANY
Com-ID	Att. of COMPANY
Com-Name	Att. of COMPANY
Com-Revenue	Att. of COMPANY
COMPANY	Entity type
manufactures	Rel-type
Pro-ID	Att. of PRODUCT
Pro-Name	Att. of PRODUCT
PRODUCT	Entity type

Figure 2.11 - The **Text sorted** view of a schema

Two important properties

- Objects that are selected (in white on black) in a view still are selected in any other view in which they appear. For instance, an attribute with a particular name can be retrieved in a schema by using the *text sorted view*. Now, choosing the *standard graphical view* allows us to examine this attribute in its context.

2. *Reverse engineering* can briefly be described as the converse of what we did in the first lesson, that is *recovering the conceptual schema of an existing database*. It involves complex techniques and tools that are described in other documents but that will be ignored in this tutorial.

- Building a schema, or examining, deleting and modifying its components, can be performed whatever the view in which this schema is displayed. For instance, double-clicking on the line of an object in a text view opens the same property box as in a graphical view.

2.5 Application: far jumps through a graphical schema

Navigating through a large schema can be fairly tricky. Let us examine the simple following problem: *considering entity type COMPANY, show the entity type that plays **the other role** in rel-type manufactures.*

In the schema of Figure 2.4, the problem is solved simply by positioning the schema window on entity type COMPANY, then in selecting the other end of rel-type manufactures.

Now, let us consider that these objects are parts of a large schema, in such a way that both entity types are **more than one meter apart** (Figure 2.12). Retrieving rel-type manufactures, then entity type PRODUCT gets much more difficult. We have to activate the scroll bars in several directions, following very carefully the arcs that make the rel-types. It is easy to confuse two crossing arcs, and to follow the wrong direction.

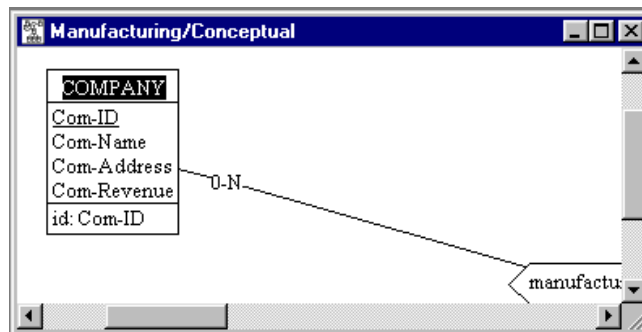


Figure 2.12 - How to retrieve the other entity type of manufactures?

The solution is to use the right view(s) in the right way:

1. in the Graphical standard view, we select entity type COMPANY (Figure 2.12);
2. we switch to the Text extended view and we click with the right button on the role line "role: [0-N] in manufactures", which sends us to rel-type manufactures (Figure 2.13);
3. in the list of roles of manufactures, we identify the opposite role ([1-1]: PRODUCT) and we click on it with the right button;
4. this sends us to entity type PRODUCT (Figure 2.14);
5. we switch to a graphical view, which shows the selected object³ (Figure 2.15), and *voilà!*

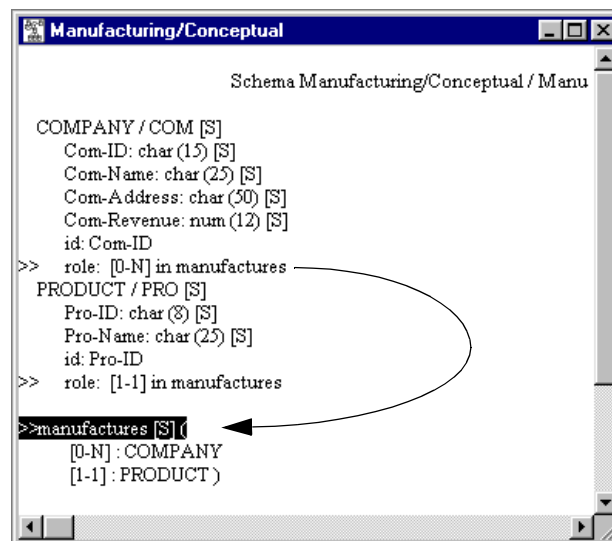


Figure 2.13 - We reach rel-type manufactures by right-clicking on the role of COMPANY.

-
3. When selected objects are outside the current window, just press the Tab key to move the window to the next selected object.

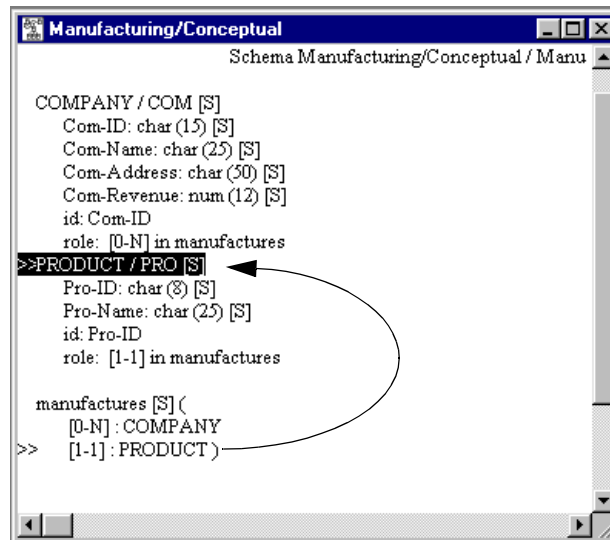


Figure 2.14 - The opposite entity type has been found ...

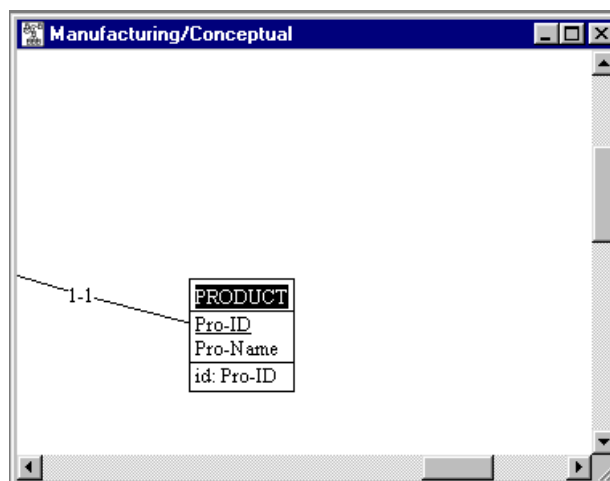






Figure 2.15 - ... and presented in a graphical view.

Key ideas of Lesson 2

1. A CASE tool can memorize the products (schemas and generated texts) of a project on secondary memory. They can be opened later on.
2. A fragment of a schema can be incorporated into a text document either as a bitmap image (through PrintScreen key) or, better, as a vector-based drawing.
3. A schema can be displayed under various formats, either graphical or textual. Each of them shows some or all aspects of the schema objects.
4. The graphical formats are more intuitive, and show the direct environment of an object.
5. The textual formats are more concise, and can show syntactic patterns of the object names.
6. Some textual formats make it possible to jump to distant linked objects.
7. Switching between different formats allows us to quickly navigate through rel-types in very large schemas.

Summary of Lesson 2

- In this first lesson, we have studied some important concepts:
 - graphical views of a schema: compact, standard
 - text views of a schema: compact, standard, extended, sorted
 - navigation through the objects of a schema

- We have also learned:
 - to open an existing project:
 - Project / Open project** 
 - to open an existing schema
 - to include fragments of a schema into a text:
 - Edit / Copy graphic** 
 - to select a schema presentation format:
 - View / Text compact**
 - View / Text standard** 
 - View / Text extended**
 - View / Text sorted**
 - View / Graph. compact**
 - View / Graph. standard** 
 - to give graphical objects rounded corners and shades:
 - View / Graphical settings**
 - in a text view, to navigate *from entity type to rel-type* and *from rel-type to entity type*: right button on the role line
 - in a graphical view, to get the next selected object in the center of the schema window: tab key

Exercises for Lesson 2

Finding interesting exercises for such a lesson is quite a challenge! If you insist, try these; otherwise start the next lesson.

Open the LIBRARY project (or its French equivalent BIBLIO) and its conceptual schema `Library/Conceptual`.

- 2.1 Examine the semantic description of the objects in the schema. Change and complete some of them.
- 2.2 Change the position of some attributes and roles in text views. Examine the graphical view and change the position of some objects.
- 2.3 Find the other side of a rel-type from an entity type.

Lesson 3

An even closer look at schemas

Objective

In this sequel to Lesson 2, we study how to manipulate graphical and textual objects, how to change their apparent or actual size, how to navigate through a schema and to generate reports. We also examine various techniques to inspect the objects of a schema.

Preliminary checking

Make sure that the project MANU-1 (file manu-1.lun) created in Lesson 1, and LIBRARY (or its French equivalent BIBLIO) that comes with the DB-MAIN software, are available.


3.1 Starting Lesson 3

We start DB-MAIN, we open the project MANU-1, then the schema Manufacturing/Conceptual.

3.2 Securing our work

This lesson, as well as the next ones, will lead us to perform various manipulations on the current schema, and therefore to *spoil* its initial shape and contents. Of course, when this happens we could restore the original version that has been saved on disk by **File / Open project**, but this is rather tedious, especially for large projects.

The *save point/rollback* technique is much quicker:

- **Edit / Save point** saves the state of the current schema (or button ) ,
- **Edit / Rollback** cancels the modifications carried out on the current schema since its last save point; in other words, it restores the state of the schema when the last save point was issued.

Three important rules:

1. There is only one active save point for the schema, but each schema of the project can have its own independent save point.
2. A save point cannot restore a schema that has been deleted (use **File / Save as** instead).
3. When a project is closed, all its save points are lost.


3.3 Manipulating the graphical components of a schema

The position of the objects of a schema can be changed by *selecting and dragging* them in the usual way. Several objects can be selected (or deselected) by pressing the `shift` key when selecting, or by drawing a selection rectangle with the mouse, and moved simultaneously.

Moving objects

Moving objects in their window obeys the general Windows rules:

- selected objects are moved by dragging them in the window space;
- selected objects are moved by pressing the cursor keys (`←` `↑` `→` `↓`);
- small-step moves are obtained by pressing the cursor keys while pressing the `Ctrl` key;
- using the scroll bars moves the window in the four directions.

The *Move mode* designates the way DB-MAIN reacts when an object is moved on the screen: does it move the object only (*independent mode*), or does it reposition the connected objects as well (*dependent mode*)? This mode can be set either in the *Graphical settings panel* (`Independent` button) or through the `INDEP.` button on the *Graphical tools bar*: .

In the *Dependent mode*, the graph is adjusted as follows (Figure 3.1 left):

- when an entity type is moved, its relationships types and their roles are moved proportionally and redrawn;
- when a relationship type is moved, its roles are moved too,
- when a role is moved, nothing else is redrawn.

In the *Independent mode*, the graph is adjusted as follows (Figure 3.1 right):

- when an object (entity type, relationship type, role) is moved, nothing else is redrawn, except the arcs that link it to the other objects.

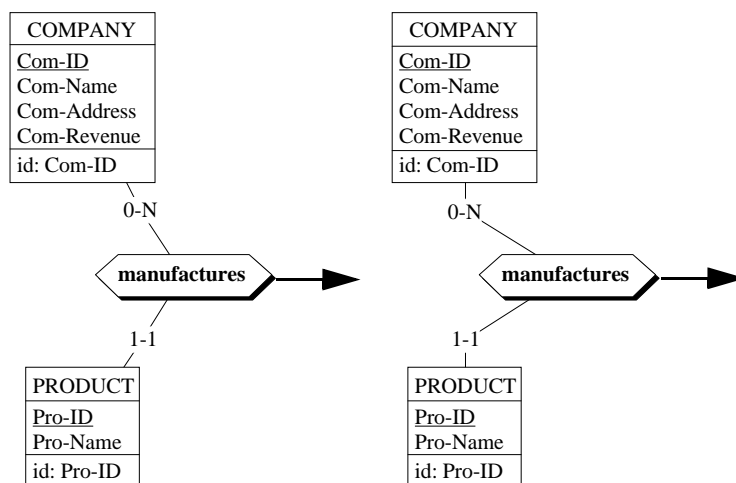


Figure 3.1 - Moving rel-type manufactures in the *dependent mode* (left) and in the *independent mode* (right).

Aligning objects

After a while, a schema may look like spaghetti, and we might want to put some order among its components. A first nice feature is the rel-type **Align** action which allows us to align a role or a relationship type according to its connected objects. We can get this effect by clicking on the object (role or rel-type) with the *right button* of the mouse (Figure 3.2).

To align a larger set of objects, we will make use of the **View / Alignment** command, that provides us with eight operators, four for vertically aligning the objects and four for horizontal alignment. They are also available on the *Graphical tools* bar (Figure 3.3).

In the **horizontal** dimension, we can align objects on their left side, on their right side, we can center them and we can distribute them horizontally at equal distance.

In the **vertical** dimension, we can align objects on their top side, on their bottom side, we can center them and we can distribute them vertically at equal distances.

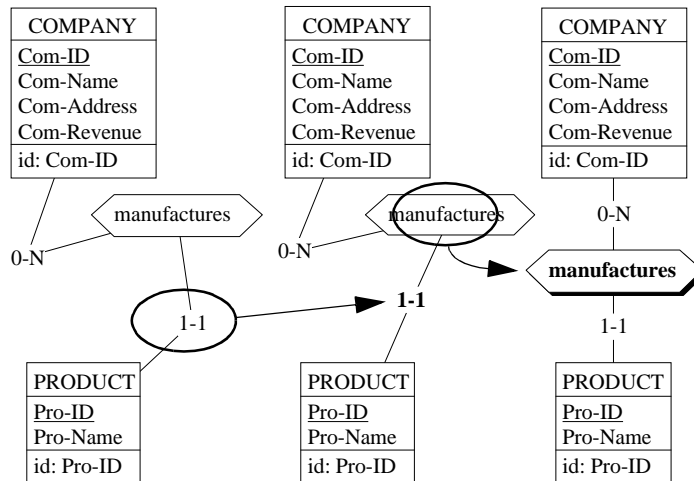


Figure 3.2 - Aligning roles (center) and relationship types (right) by clicking with the **right** button of the mouse.

Horizontal object moves	
	: align to left
	: align to right
	: center horizontally between left and right
	: distribute evenly between left and right
Vertical object moves	
	: align to top
	: align to bottom
	: center between top and bottom
	: distribute evenly between top and bottom

Figure 3.3 - The eight object alignment operators.

Two comments:

1. *Horizontal* means that the objects are moved *horizontally* to reach their final position (the same for the *vertical* direction).





Arc alignment	
	: horizontal staircase
	: vertical staircase
	: top corner
	: bottom corner

Figure 3.4 - The four arc alignment operators.

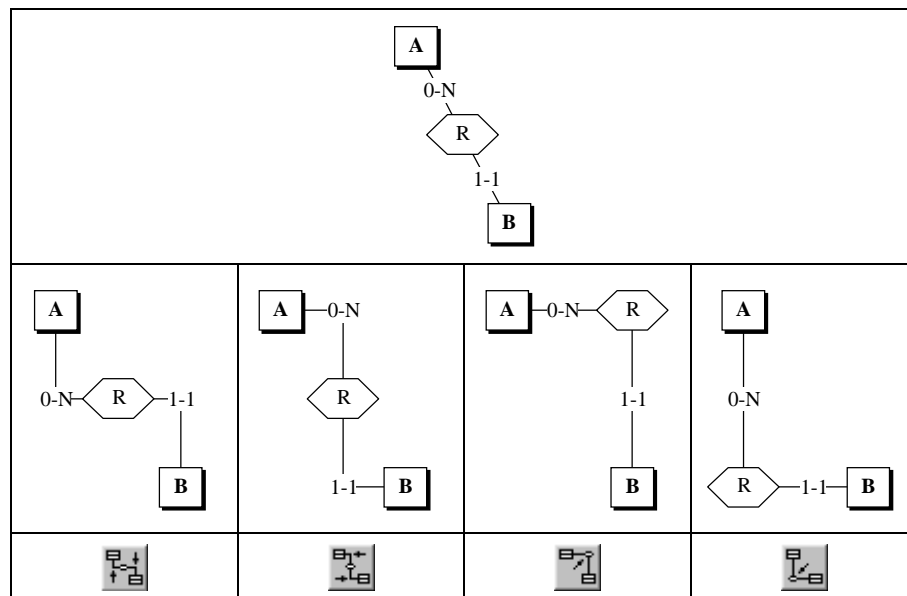


Figure 3.5 - How to draw a source rel-type (top) with *staircase* style with just a mouse click.

- When the objects are distributed evenly, the distance is evaluated between the edges of the objects, not between their centers. This provides a natural positioning of roles and rel-types between their entity types.

The last four alignment operators (Figure 3.4) are dedicated to users who are fond of *staircase* rel-types. Since an image is worth one thousand words, we suggest you had a look at Figure 3.5.

The best way to get acquainted with these operations is to play with a disaligned schema such as that of Figure 3.6, which is available in project `Manu-3.1un`, schema `Alignment`.

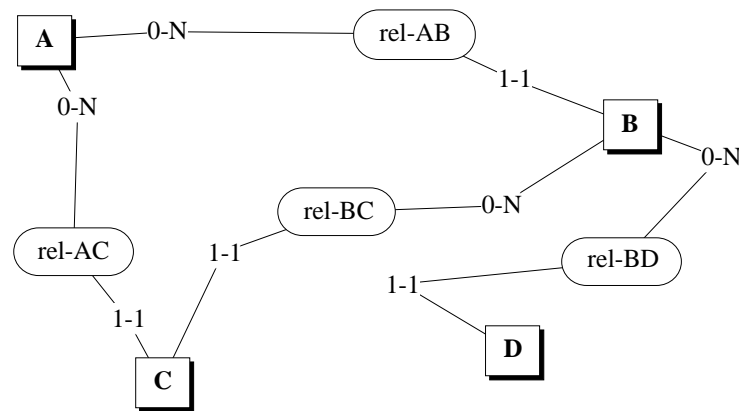




Figure 3.6 - This schema obviously suffers from a severe disalignment disease. Cure it.

Zooming in and out

For large schemas, a *zooming function* is available to help fit a larger or a smaller portion of the schema in the Schema window (zoom out), or to examine tiny details (zoom in). This function is available in the *Graphical settings panel* (Figure 2.5) and in the *Graphical tools bar* (Figure 2.1) through the following buttons:

-  expands the schema representation by 10%;
-  shrinks the schema representation by 10%;



sets the zoom factor by specifying its exact value; the `fit` value adjusts the zoom factor so that the schema fits in the schema window.

Fonts

The **font**, **font size** and **style** of the object names can be changed through the command **Edit / Change font**. You can get a more compact view by decreasing the character size.

Grids


Organizing large schemas may require the use of the **Grid function**, which draws lines that decompose the graphical space into equal size pages (see the *Grid format* block in the *Graphical settings* panel, Figure 2.5). The page size can be standard (*A3*, *A4*, *Letter*), with portrait or landscape orientation, compliant with the current printer, or customized.

The Reduce function


This is a way to change (shrink or expand) the actual size and position of each object of the current schema by a certain factor. It seems similar to the *Zoom* function, but the latter only defines how close you are from the schema, while leaving the objects themselves unchanged. The following table should make the differences between Reducing and Zooming more explicit.

	objects	object appearance	grid	grid appearance
zoom out by 50%	unchanged	reduced by 50%	unchanged	reduced by 50%
reduce by 50%	reduced by 50%	reduced by 50%	unchanged	unchanged

Colors

Normally, all the objects in a schema are drawn, and their names are written, in **black**. You can change this by selecting objects, then giving them **another color**. Use command **Edit / Color selected** or click on the  button in the *Standard tools* bar to give them the current color. This color can be changed by the command **Edit / Change color**.

Marking objects

This is a very simple and powerful means to define persistent subsets of objects in a schema. *Marking* objects is obtained by first selecting the objects, then asking **Edit / Mark selected** or clicking on the button  in the *Standard tools* bar. To *unmark* objects, just mark them again. Marked objects are drawn with specific attributes (Figure 3.7):

- a marked *entity type* is shaded (unless it was already shaded, in which case it is *unshaded*) and its name is written in boldface;
- a marked *rel-type* is shaded (unless it was already shaded, in which case it is *unshaded*) and its name is written in boldface;
- a marked *attribute* is written in boldface;
- a marked *identifier* (or group) is written in boldface;
- a marked *role* is written in boldface.

The main difference between *selecting* and *marking* objects is that marking is a permanent state while a selection is volatile. Closing a schema and saving it on disk keep all the marks until we change them explicitly.

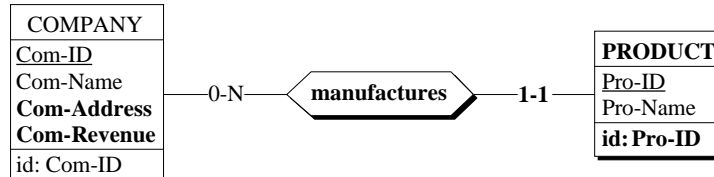


Figure 3.7 - Entity type **PRODUCT**, rel-type **manufactures**, attributes **Com-Address** and **Com-Revenue**, role **PRODUCT** of **manufactures** and identifier **Pro-ID** are marked.

Retrieving all the marked objects in a schema cannot be simpler: just select them by **Edit / Select marked**. To *unmark all the marked objects* of a schema, select them as above, then mark them again.

In fact, it is possible to define up to 5 different sets of marked objects. Such a set corresponds to a **marking plane**. All the marking operations are carried out in the current marking plane. Changing the current plane is done through a special button in the *Standard tools* bar:



When a combination of marked and unmarked objects are set to *marked*, the result is that they are all marked. This makes it possible to combine the objects marked in two planes into a third one:

1. *transfer from plane 1 to plane 3*: choose the first plane, select the marked objects, choose the third plane and mark the selected objects,
2. *transfer from plane 2 to plane 3*: choose the second plane, select the marked objects, choose the third plane and mark the selected objects.

If no objects are marked eventually, this means that planes 1 and 2 were identical. In this case, just do step 1 again.

There are numerous applications of marking planes:

- in a large schema which is still in a validation phase, objects which are already checked are marked; the schema is completed when all objects are marked;
- objects which have been given a semantic description are marked; the schema is completely documented when all the objects are marked;
- in a schema that comprises objects from different sources, each source is marked in a different marking plane; an object can be marked in more than one plane¹;
- marked objects can be manipulated by DB-MAIN processors (as we will see later on)
- some DB-MAIN processors can return marked objects (as we will see later on).

Auto-draw

If the layout of a schema does not fit your taste, you can ask DB-MAIN to suggest a better spatial arrangement through the command **View / Auto-draw**. This function is particularly useful for large schemas that have not been entered graphically (through the *reverse engineering* extractors for instance)². If *Auto-Drawing* a schema does not produce a satisfying result, try *auto-drawing* again. Generally, you will need to fine-tune the layout manually.

-
1. For this aim, a *view schema* is a much better means to define an arbitrary number of subsets of objects. They will be studied in Lesson 12.
 2. This function is at its best for large and complex schemas. It does not provide satisfying results with small schemas. Before using it, it can be wise to save the schema state through the **Edit / Save point** function, then to restore this state by **Edit / Rollback** if the result is not satisfying.

Using a larger schema

To get a better feeling of the usefulness of the various views, we switch to another project. We close the current one (command **File / Close project**), and we open the `LIBRARY` project (command **File / Open project**) and its conceptual schema. Now we experiment with each view, and try to figure out the meaning of the components of this schema, which obviously describes the management of a scientific library. Its contents include many more modeling characteristics that will be discussed later.

Last observations

We observe that:

- switching from a view to another one is immediate, and can be asked for at any time;
- the operations of the tool are independent of the view through which they are executed;
- an object that is selected (highlighted) in a view still is selected when we switch to another view;
- if several schemas of a project are opened (more on this later on), they can be displayed in different views.

3.4 Navigation through textual views

When a schema is small, it spans one or two screens only. Retrieving an object in such a schema needs no special skill nor any special tool. The problem is less trivial when the schema is larger, and is several dozens of screens large (large schemas can include thousands of entity types and rel-types): browsing through such a schema can be time consuming and does not guarantee that the objects we are looking for will be found quickly, if ever.

Retrieving a specific object can often be made easier by working first on the **Text compact** and **Text sorted** views, using them as some kind of dictionaries, then switching to the standard graphical or text views when the object of interest has been found.

Another useful tool for object retrieval in context is the navigation feature of `DB-MAIN`. To illustrate them, we need a larger schema, such as `LIBRARY`.

We display it in the **Text extended view**, and we reduce the Schema window a little bit to simulate a *large schema in a too small window*.

Let us experiment the navigation capabilities of DB-MAIN. Unless told otherwise, the following manipulations are valid for the **Text standard** and **Text extended** views.

- We select the COPY entity type by clicking on its name; we observe that each line in which the name COPY appears (i.e., each instance of COPY) is tagged with symbols ">>"; such is the case for each role in which COPY appears;
- If we press the TAB key; the next tagged instance of COPY appears in the center line of the Schema window; this allows the cursor to jump to each of the relationship types in which COPY takes part;
- We click with the *right button* on a line describing a role in which COPY appears, in a rel-type paragraph; the COPY entity type is then selected; the right button acts as a *go home* button;
- In the **Text extended** view, we click with the *right button* of the mouse on a role in which COPY appears, in its entity type paragraph, then click; the relationship type of the role is then selected

In Figure 3.8, the navigation rules are shown on the small project Manu-1.

3.5 Reordering attributes and roles

Though the order in which attributes (and roles) appear in the textual and graphical views does not matter in most situations, you may want to change this order.

To **change the position of an attribute** (graphical and text views), select it, then

- press the Alt + ↑ keys³ to move it one position up,
- press the Alt + ↓ keys to move it one position down (Figure 3.9).

To **change the position of a role** (text views), select it, then

- press the Alt + ↑ keys to move it one position up,
- press the Alt + ↓ keys to move it one position down.

3. The keys must be pressed simultaneously, not sequentially.

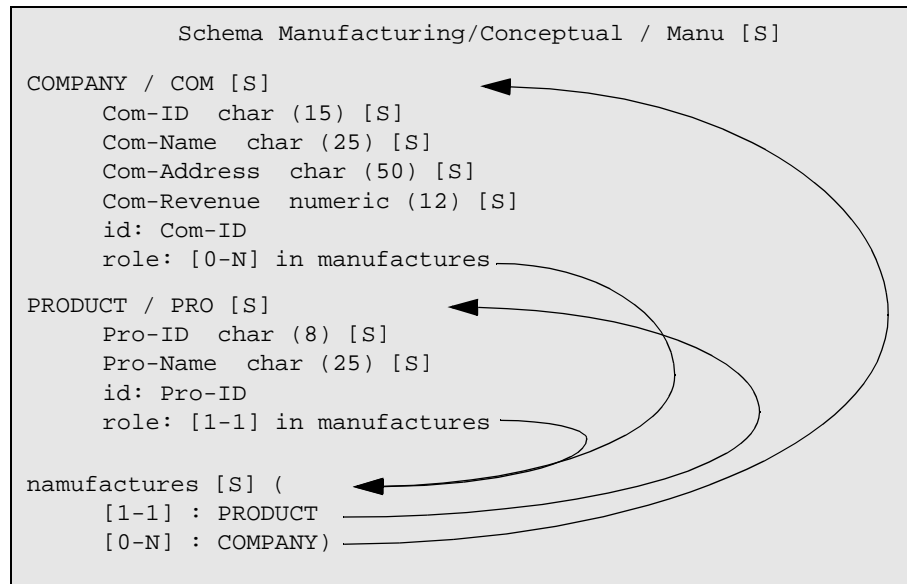


Figure 3.8 - Navigating in the **Text extended** view of a schema of project Manu-1 with the *right button* of the mouse.

There are other ways to reorganize the attributes of an entity type, but they require more sophisticated functions (namely schema transformations) that will be studied later.

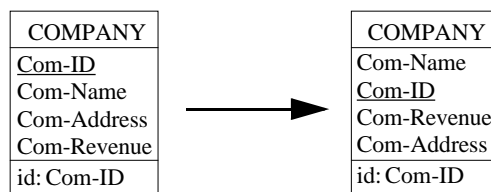


Figure 3.9 - Changing the order of the attributes with Alt + ↓↑.

3.6 Generating reports

A decent CASE tool must produce external documents that can be printed on paper. This one does it too. Several kind of reports can be of interest, ranging from simple object lists to sophisticated documents including a table of contents, an index and footnotes. Though DB-MAIN can produce such documents, we will show how to generate simple outputs. Three formats are available from the menus through the command **File / Report**.

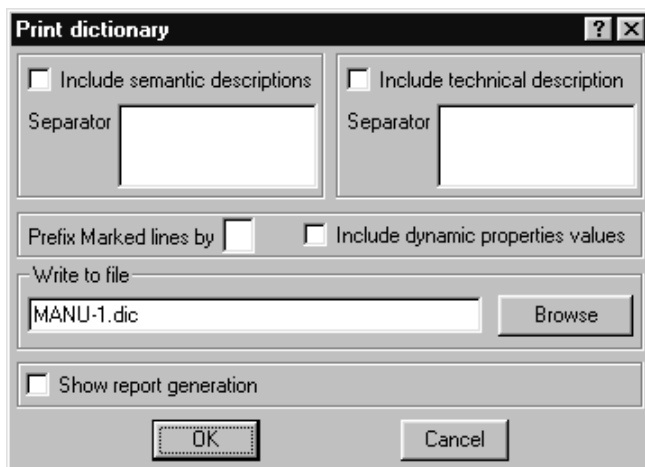



Figure 3.10 - Generating a simple text report.

1. **Textual view**: the schema must be displayed in a *text* view (for instance with button ). Its representation is sent to a file with some format options that are set through the *Print dictionary* panel (Figure 3.10). This function produces a *.dic plain ASCII file.

This panel allows us to specify

- the output file,
- whether we want the semantic description to be included,
- what character string will be included just before each semantic description (a *tab* control can be used to clearly separate it from the object description⁴),

4. According to the Windows conventions, a **tab** control is entered as *Ctrl + Tab*.

- how the lines of marked objects are tagged.

We will ignore the other parameters for now⁵.

Formatting the output text with a text/document processor can provide reports such as that of Figure 3.11.

2. **RTF**: the schema is saved as a formatted RTF document. Various options are available⁶.
3. **Custom**: the schema is processed through a customized Voyager 2 program⁷.

For immediate needs, you can directly send the current schema to the printer, be it in graphical or textual view, through command **File / Print**. The printer can be chosen and configured through **File / Printer setup** as usual.

There are other ways to produce reports. Let us remember one of them: the *Copy graphic* function, that allows us to include fragments of schemas into standard texts (Section 2.2).

3.7 Copying objects

When building a schema, it can happen that several entity types have to be given similar attributes, or that the schema includes parts that are almost the same. Instead of entering the similar objects manually, it could be more convenient to copy the original fragment, then to modify the copy.

The procedure is as expected:

1. select the components to copy and put them on the clipboard (ctrl+C or **Edit / Copy**);
2. paste them in the schema (ctrl+V or **Edit / Paste**);

-
5. For those who **do** want to know: Include dynamic peoperties values means that user-defined object properties are to be included in the report (see Lesson 16) and Show report generation means that we want the report and its generation process to appear in the project windows (in fact in the process history). In this lesson, the effect of checking this button will be to show the report as a product in the project window.
 6. Check that the DB-MAIN directory includes the files **XXXXXX** and **XXXXXX**. You can define their paths through **File / Configuration**.
 7. We will tell some words about these programs later on. For now, it suffices to know that Voyager 2 is the programming language of DB-MAIN allowing its users to develop their own functions.

Dictionary report	
Project	MANU-1
<hr/>	
	Schema Manufacturing/Conceptual
	A simple example of conceptual database schema used in the first lessons of the DB-MAIN tutorial. This schema has been created on December 15, 1998.
* COMPANY	A registered business organization with which we have had commercial contacts for less than 5 years.
Com-ID	Internally assigned company Id.
Com-Name	Official name of the company.
Com-Address	Main address of the company.
Com-Revenue	The total net income of company for the last fiscal year.
id: Com-ID	
* PRODUCT	A product of interest for our company.
Pro-ID	Internally assigned product Id.
Pro-Name	The conventional name of the product.
id: Pro-ID	
* manufactures (Specifies which products are manufactured by each company.
[0-N] : COMPANY	
[1-1] : PRODUCT)	

Figure 3.11 - A simple text report.

- if the the pasted objects are attributes, first select an entity type, a rel-type or an attribute; the pasted objects will be inserted after this insertion point (Figure 3.12).

If needed, DB-MAIN makes the names of the pasted objects unique through the addition of a small suffix.

Note. Copying attributes to define near similar objects can be an evidence of a more complex situation, where entity types appear to be subtypes of each other, or to have a common supertype. These structures will be studied in Section 8.2. v

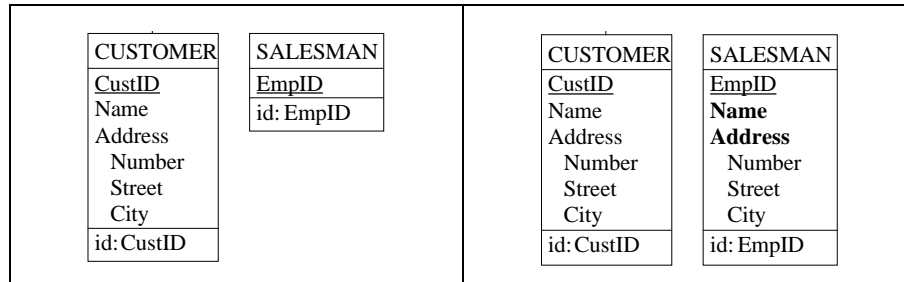



Figure 3.12 - It appears that SALESMAN must be given attributes similar to Name and Address of CUSTOMER (left). Select the latter, type ctrl+C, select EmpID of SALESMAN then type ctrl+V (right).

3.8 Inspecting objects

Examining the properties of the objects in a schema, or even the schema or the project themselves, is quite easy. We double-click on the object (in any text or graphical view) and its property box opens. To read its semantic description, we click on the Sem. button, which opens the semantic description window.

If we have to examine a large number of objects, this procedure may appear tedious, and even painful. There exist two quicker ways to inspect the properties of the objects.

First, opening the semantic description window can be done by selecting the object and clicking on the SEM button in the Standard tools bar .

The second way is much more powerful, and uses a new DB-MAIN feature called the *Property box*. It is opened through command **Window / Property box**, and appears as in Figure 3.13.

This box is permanent until it is explicitly closed. It shows in real time all the properties of the current object, i.e., the object which is currently selected. Simply select another object, and the box shows the properties of this object.

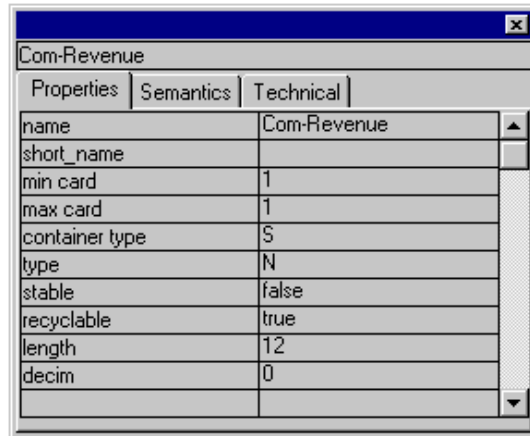


Figure 3.13 - The contents of the *Property box* when attribute Com-Revenue is selected.

The Property box has three panels. The second one shows the semantic description of the selected object (Figure 3.14). Now, reading the semantic description of a series of objects can be done by merely selecting each of these objects.

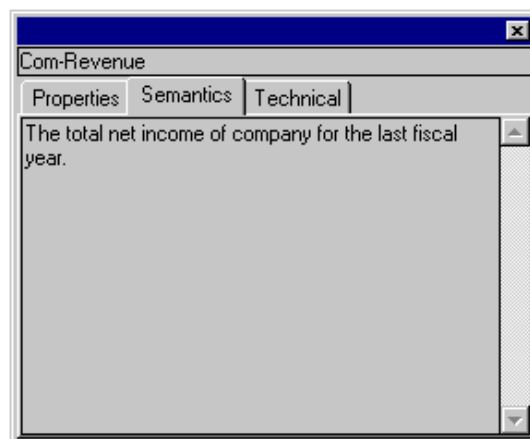


Figure 3.14 - Examining the Semantic description of an object through the Property box.

Note. The Property box is read-only, and cannot be used to update the properties of an object. To do so, we must use its standard Object property box. v

3.9 External links

In some cases, the documentation of an object (entity type, attribute, etc.) is available in external documents that cannot be included in the project. So, all we can do is to reference these external documents from the schema objects, and ask the users to go and read these documents.

In any semantic description, typing the name (and correct path) of an external document has an interesting side effect: this name is an external link to the original document. Therefore, double-clicking on this name opens the document with its source processor (Figure 3.15). Provided your system knows the associations between the file extensions and the processors, just double-clicking on a document name allows you to read a Word document, examine a PDF text, view a video movie or get a web document.

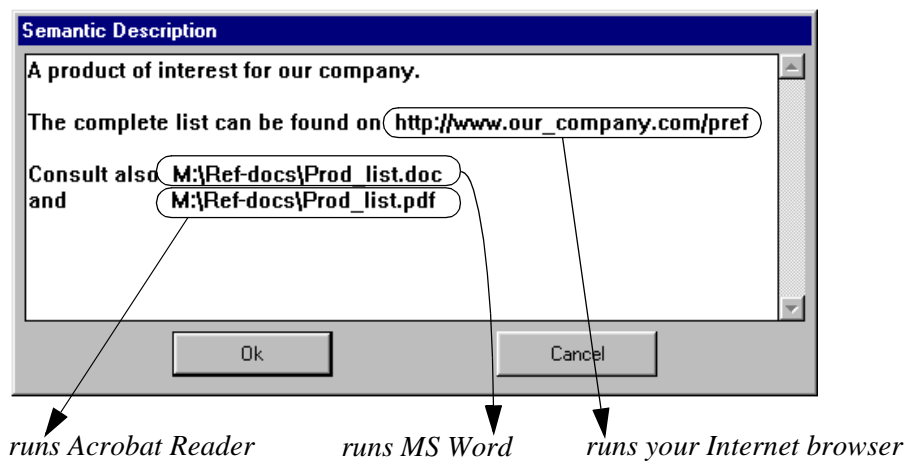


Figure 3.15 - Automatic access to external documents.

3.10 Quitting the lesson

We will still use this project later on. Therefore, we save it with the name `manu-3.lun` and we quit DB-MAIN.

Key ideas of Lesson 3

1. A CASE tool must allow its users to **cancel all the modifications** since a given reference point (save point).
2. One of the main functions of a CASE tool is to offer an easy-to-use **graphic editor** that allows the developer to draw objects, to move objects, to align objects, to enlarge or reduce the viewing angle (zoom), to change the size of objects (reduce), to distribute objects into pages (grid), to color objects, to reorder objects, to experiment with various schema layouts (auto-draw).
3. In a large schema, persistent and dynamic **subsets of objects** can be defined and combined (mark, marking planes).
4. The various text views of a schema complement the graphical views. They provide **hyperlink** capabilities to **navigate** through the objects of the schema.
5. External documents of any sort can be mentioned in a semantic description. Their names form active hyperlinks to these documents
6. A CASE tool can produce **printed** and **formatted documents** or *reports* of various kinds.
7. Objects have **specific properties** (name, type, semantic description, etc.) that can be inspected easily (specific property boxes, the general Property box).

Summary of Lesson 3

- In this lesson, we have studied the following concepts:

- save point of a schema
- graphical aspects of a schema (zoom and reduce)
- marking planes
- text navigation through *role* links
- reordering attributes and roles
- simple reports
- general property box
- active external documents.

- We have learned:

- to define a save point for a schema
- to cancel schema modifications
- to move objects in the schema
- to change the move mode of objects
- to align rel-types and roles
- to align a set of objects

Edit / Save point (or button )

Edit / Rollback

{← ↑ → ↓} and Ctrl + {← ↑ → ↓}

View / Graphic. settings 

right button of the mouse

View / Alignment



- to zoom on a schema in and out

View / Graphic. settings



- to reduce or expand a schema
- to change the font of a schema
- to draw a grid in the schema space
- to color schema objects
- to change the current color

View / Graphic. settings

Edit / Change font

View / Graphic. settings

Edit / Color selected 

Edit / Change color

- to mark/unmark objects and to play with marked objects

Edit / Mark selected



Edit / Select marked

- to select a marking plane



- to ask DB-MAIN for a new schema layout **View / Auto-draw**
- to retrieve instances of an entity type in a text schema

tagged lines *and* **tab** key

- to navigate between entity types and rel-types in a text schema
right button of the mouse
- to change the order of attributes and roles in an entity type
alt + ↑ ↓
- to copy selected objects elsewhere in the schema or in another schema of the project:

Edit / Copy (ctrl+C)

Edit / Paste (ctrl+V)

- to generate simple text reports

File / Report / Textual view

- to generate sophisticated reports

File / Report / RTF

- to generate custom text reports

File / Report / Custom

- to print a schema on the printer

File / Print

- to choose and configure the printer

File / Printer setup

- to inspect objects quickly

Window / Property box *and*

- to access an external document from the semantic description of objects:

double-click on the document name

- We have produced a new type of file:
 - dictionary reports (*.dic).

Exercises for Lesson 3

- 3.1 Open project `Library` (or its French equivalent `BIBLIO`) and schema `Library/Conceptual`. Generate and print a report based on each of the text views. Try to find specific uses for each of them.
- 3.2 Open a Text standard report with a text processor. Include after each entity type title the graphical representation of the entity type (through the **Copy graphic** command).
- 3.3 **Aligning objects.** I'm not quite sure that you have completed the exercise suggested in Figure 3.6! Now it's time to do it.
- 3.4 **Zooming and reducing.** Open project `Library` and schema `Library/Conceptual`.
Define a grid based on the A4/Landscape or Letter/Landscape paper format.
Choose a zoom factor such that the current page fits in its schema window.
Choose the maximum reduce factor such that the schema still fits into the current page. Print it to check.
- 3.5 **Marking and coloring objects.** Use plane 1 to mark the entity types, plane 2 to mark rel-types and roles, plane 3 to mark the attributes and plane 4 to mark the identifiers.
Transfer the marks of planes 1, 2, 3 and 4 to plane 5.
Color each plane in a different color.
Unmark all the planes and color all the objects in black.
- 3.6 Examine the semantic description of all the objects of schema `Library/Conceptual` **in less than 1 minute**.
- 3.7 Type the name of a document in the semantic description of an entity type. Open this document from `DB-MAIN`.

Lesson 4

Multi-product projects

Objective

This lesson introduces the concept of *multi-product projects* by considering the example of a design in which we distinguish the conceptual schema and the logical schema of a database as well as two text files. Some characteristics of relational logical schemas are examined. Additional functions related to schema and object management are described as well. Transfer of components between projects is described through export/import functions.

4.1 Starting Lesson 4

We start DB-MAIN, we open the project MANU-3, then the schema Manufacturing/Conceptual.

4.2 Conceptual and logical schemas

The way we worked in Lesson 1 to produce an SQL database structure was a bit simplistic: we designed a conceptual schema, then we generated the equivalent SQL code to be executed by an RDBMS¹. This procedure is fine for small databases, but is not realistic for large projects. Of course, it is much too early to tackle the problems induced by managing complex projects, but we can already introduce the concept of *multi-schema projects*, i.e., projects that include more than one schema, through a more sophisticated procedure than that suggested in Lesson 1.

Let us suppose that we want to keep in the project not only the description of the *conceptual schema* (i.e., the current schema Manufacturing/Conceptual), but also the description of the *logical schema*. In traditional database design methodologies, the logical schema is intended to describe the same real-world situation as the conceptual schema does, but in technical terms of tables, columns, primary keys, foreign keys and indexes instead². The logical schema is made up of the database structures that are encoded into a SQL program.

To develop these concepts, we need to go back to the project Manu-3 that is currently opened.

To give us the opportunity to go through this lesson again later on, we work on a new project called, say, Manu-4, which has the same contents as Manu-3, at least initially.

To do so, we call the Project property box through the command **File / Project properties**, we modify the name into Manu-4, and save the current project (**File / Save project as**) as Manu-4.lun. From now on, we have two projects,

-
1. Relational Database Management System. Sybase, Informix, Oracle, SQL Server and Access are some examples of RDBMS.
 2. See the lessons of Volume 2, or reference textbooks such as [Teorey,1998], [Batini,1992] or [Blaha,1998].

namely Manu-3, which is closed and Manu-4, the current project on which we will work. So far, these projects have the same contents.

Building a *relational logical schema* is fairly easy, though we may have no idea on how to translate a conceptual schema into relational structures, i.e., into tables, columns, keys and the like. Indeed, DB-MAIN proposes a function which carries out this translation automatically by replacing a schema by its SQL logical equivalent version. Since we want to keep both schemas in the project, we proceed as follows:

1. *Cleaning and modifying the Manu-4 project.*

We can get rid of the schema `Alignment`, that is no longer useful. In the same way, we delete the SQL program generated in Lesson 1.

Deleting objects is quite simple and intuitive: we select the objects, then we press the Del key. Another way is through the command **Edit / Delete**.

Now, the project looks like Figure 4.1.

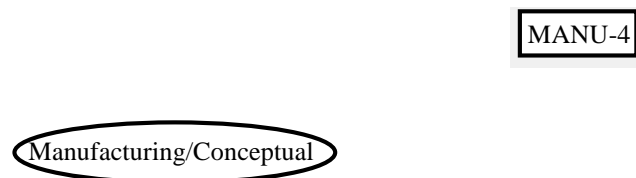


Figure 4.1 - The Manu-4 project in its beginning state.

Our conceptual schema is a bit simplistic, and we could find it interesting to enhance it a little. We open the schema, and we state that a *product can be manufactured by an arbitrary number of companies*. Accordingly, we change the cardinality of the role `manufactures.PRODUCT`³ from `[1-1]` to `[0-N]`⁴. To do so, we double-click on the role and we change the cardinality value, either by typing it or by selecting it in the listbox. The new version should appear as in Figure 4.2.

-
3. A role can be designated by the name of the rel-type followed by the name of the entity type. Another way to denote roles will be seen later.
 4. As will be observed, the foreign key of Lesson 1 will be replaced with a connection table.

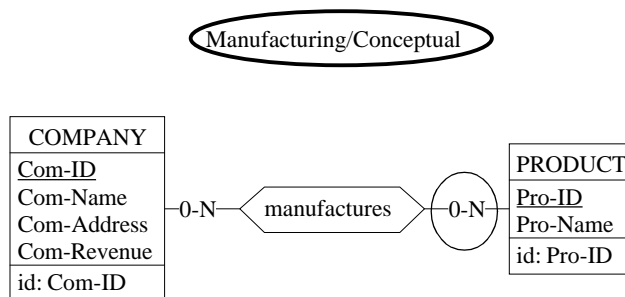


Figure 4.2 - The new Manufacturing conceptual schema.

2. *Making a copy of the first schema.*

Let us make a copy of the conceptual schema:

- we select the source schema in the Project window, or we open it (it is the current case);
- we execute the command **Product / Copy product**;
- the *Schema property box* opens and proposes default characteristics for the new schema: the name is that of the source schema, "Manufacturing", while the version proposed is "Conceptual-1". We change the version into "Relational" and we click on the button OK.

The project window shows the new schema as well as its relationship with the source conceptual schema (Figure 4.3).

We open the so-called *Relational* schema. Not so surprisingly, it includes the same objects as the conceptual schema, which is fairly common with copies!

3. *Translating this copy into relational structures.*

Now we will transform this schema into relational structures. We execute the command **Transform / Relational model**. The contents of the windows are replaced by SQL structures. To improve the readability, we shade the "entity types" (through **Views / Graphical settings**), now to be interpreted as tables. If things have gone right so far, the schema Manufacturing/Relational should now read as in Figure 4.4.

MANU-4

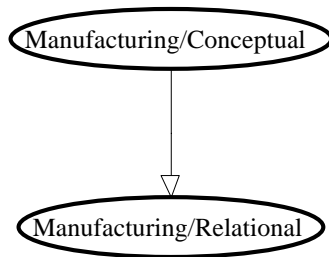


Figure 4.3 - The new *Relational schema* deriving from the *Conceptual schema*.

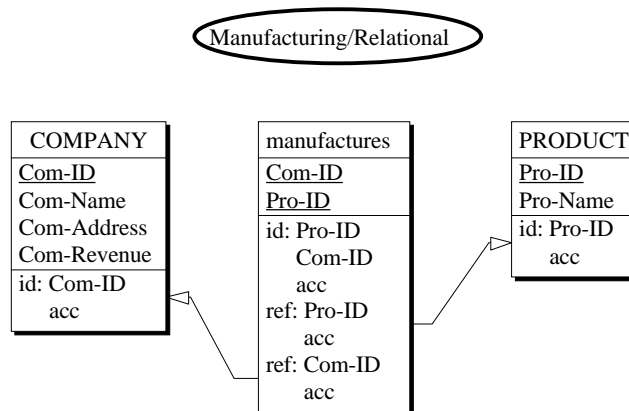


Figure 4.4 - The Relational schema.

This schema is no longer a conceptual schema since it represents data structures of a specific DBMS: each *entity type* represents a *table*, each *attribute* represents a *column* and each *identifier* represents a *primary key*. This kind of schema is called a *relational logical schema*.

The main modification of the schema is the translation of relationship type *manufactures* into entity type *manufactures*.

We observe that the table `manufactures` is made up of the column `Com-ID` which acts as a reference, i.e., a foreign key (ref), to the table `COMPANY`, and of the column `Pro-ID` which references the table `PRODUCT`. Both reference columns form the identifier (i.e., the *primary key*) of the table. In addition, an index (access key or acc in the graphical view) is defined on each identifier and on each reference column to give these structures reasonable performance. Later on, we will examine in greater detail the way identifiers, foreign keys and indexes are built and represented.

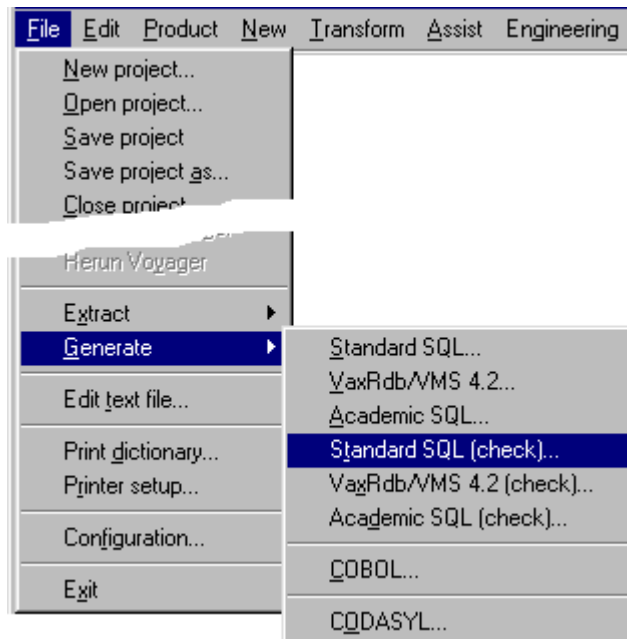


Figure 4.5 - Generating a SQL program from the Relational schema.

4.3 SQL code generation

Currently, we have two schemas in our project, but still no SQL program to build the corresponding database in the target computer. Therefore we need a final operation to generate this SQL code. We could use the command **Trans-**

form / Quick SQL as in Lesson 1, but we will explore a more professional way.

We execute command **File / Generate**, then we select the **Standard SQL (check)** style (Figure 4.6). There are other more sophisticated ways to produce SQL code, but for the purpose of this lesson, this style is quite sufficient.

```
create database Manufacturing;

create table COMPANY (
    Com-ID char(15) not null,
    Com-Name char(25) not null,
    Com-Address char(50) not null,
    Com-Revenue numeric(12) not null,
    primary key (Com-ID));

create table manufactures (
    Com-ID char(15) not null,
    Pro-ID char(8) not null,
    primary key (Pro-ID,Com-ID));

create table PRODUCT (
    Pro-ID char(8) not null,
    Pro-Name char(25) not null,
    primary key (Pro-ID));
alter table manufactures add constraint FKman_PRO
foreign key (Pro-ID)
references PRODUCT;

alter table manufactures add constraint FKman_COM
foreign key (Com-ID)
references COMPANY;

create unique index IDCOMPANY
on COMPANY (Com-ID);

create unique index IDmanufactures
on manufactures (Pro-ID,Com-ID);

create index FKman_PRO
on manufactures (Pro-ID);

create index FKman_COM
on manufactures (Com-ID);

create unique index IDPRODUCT
on PRODUCT (Pro-ID);
```

Figure 4.6 - The SQL program. The comment lines have been removed to shorten the figure.

This SQL code may not work as such on some DBMS. Indeed, some processing should have been done before generating this text. We will discuss these problems in further lessons.

4.4 Generating reports

To complete the project, we generate a report from the conceptual schema. When executing the command **File / Print dictionary** on a text view of the schema, we check the button "Show report generation" (Figure 3.10) to include the icon of the report in the *Project window* (Figure 4.7). Since any derived product is placed under its source, we sometimes have to move it to a better position.

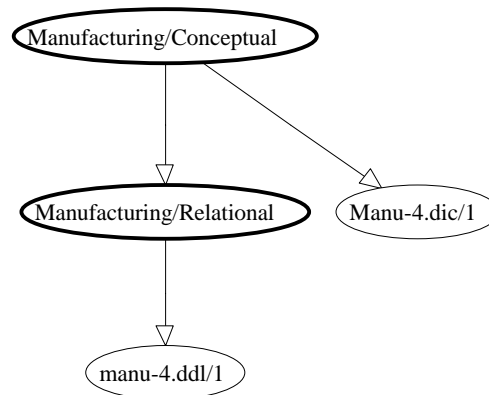


Figure 4.7 - A report has been generated from the conceptual schema.

4.5 Multi-product project

So far, our project comprises four documents or *products*, namely two schemas and two text files. A large project can include hundreds of products.

It is sometimes useful to examine two products in parallel. The best way to proceed is as follows:

- open both products,
- minimize the *Project window* (click on the leftmost of the three buttons at the top right corner on the window),
- organize the windows by **Window / Tile**.

Figure 4.8 shows the conceptual and logical schemas while Figure 4.9 presents the logical schema and its SQL equivalent side by side.

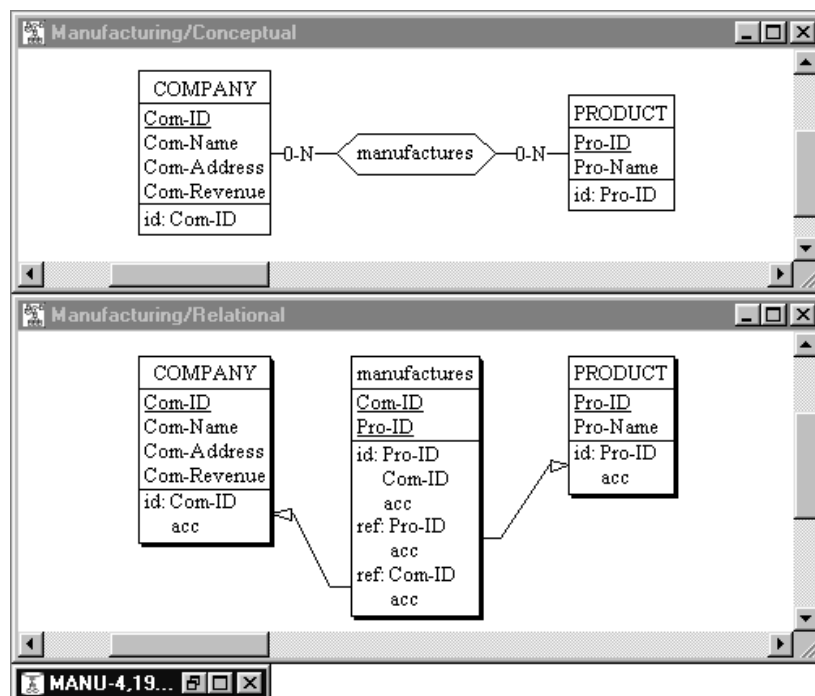


Figure 4.8 - Comparing the conceptual and logical schemas.

If we want to make the schema disappear from the screen, we can close it by closing its *Schema window*, i.e., by clicking on the close button of that window (the X button at the top right corner). Opening it again can be done by double-clicking on its icon in the *Project window* (Figure 4.7).

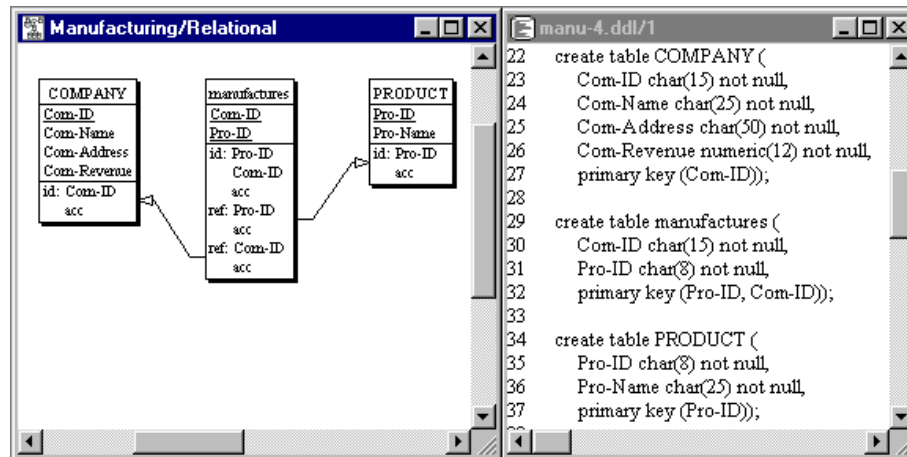


Figure 4.9 - Comparing the logical schema with its SQL text.

4.6 Deleting objects

Deleting components of a project is the simplest thing on earth: we select the objects, then we press the Del key on the keyboard. This applies to entity types, relationship types, roles, attributes, groups (e.g., identifiers), constraints and even schemas. An alternate way consists in executing the command **Edit / Delete**.

There is no way to delete a project but by deleting its *.1un file from Windows.

4.7 Export/import of schema components

Let us consider that we want to develop a new project that is fairly similar to another existing project. Quite naturally, we want to reuse some part of the specifications of the latter. Let us say that these projects share large sections of their conceptual schemas.

One way to proceed could be to take a copy of the *.1un file of the source project, then to modify the copy according to the current needs: we change the project name, we delete the unwanted products and we prune the conceptual

schema to delete the entity types, rel-types and attributes that are non relevant in the current application domain. This is a brute force approach of which we should not be particularly proud!⁵ So, we will proceed in a more refined way.

Export/import is such a better technique. Grossly speaking, it consists in identifying and selecting the relevant components of the source schema, then in transferring them to the new project.

The first step consists in **exporting** the desired specifications from the source schema:

1. Open the project `library.lun`.
2. Select the conceptual schema `LIBRARY/Conceptual` and the logical schema `LIBRARY/Logical Rel` (don't open them).
3. Execute the command **File / Export**. Accept the name suggested, namely `Library.isl`.

This procedure generates a file called `Library.isl` that includes the specifications of the exported schemas.

The second step consists in **importing** these schemas into another project:

4. Create a new project or open an existing one (that which was used in this lesson for instance).
5. Execute the command **File / Import**; choose the file `Library.isl`.
6. Select the schemas you want to include in the current project (Figure 4.10).

The current schema includes a copy of the imported schemas. If a schema to be imported has the same name as an existing one, the name of the former is changed in order to make it unique in the project.

This procedure is adequate when we want to import whole schemas. But what if we only need some objects to be imported from a source schema? In this case, the export phase is slightly different:

1. Open the project `library.lun`
2. Open the source schema (e.g., the conceptual schema `LIBRARY/Conceptual`) and select the specific objects you want to export.
3. Execute the command **File / Export**. Accept the name suggested, namely `Library.isl`.

5. Many text processing mistakes come from such brutal copy/paste techniques.

Now, the file `Library.isl` includes only one schema, comprising the selected objects. Importing these objects in the target schema is as described above.

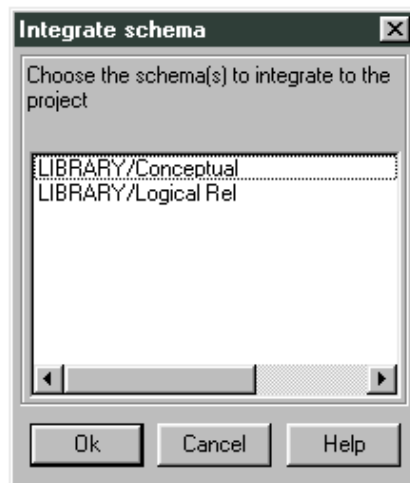


Figure 4.10 - Choosing the schema(s) to import into the current project.

Why to import schemas?

We have based the discussion on a specific goal, that is, to reuse specifications already developed in another project.

This is not the only reason why we could want to import schemas. Specification integration is another activity which requires importing schemas. We can sketch the problem as follows.

Let us assume that a large application domain has been decomposed into homogeneous subsystems, each of them being taken in charge by a developer (or by a team of developers). So, each subsystem is analyzed independently by a developer, who, eventually, produces a conceptual subschema⁶. As we can expect, any two subschemas, though they comprise different objects, will probably include some objects that model the same application concepts. For ins-

6. The term subschema is used to suggest that this schema describes a part only of the target application domain. Technically, a subschema is just a schema.

tance, the concept of *Product* will be represented in the *Production* subschema, but also in the *Warehouse* subschema. So, merging these subschemas will produce a global conceptual schema that includes one representation of each application domain concept, and therefore that encompasses the concepts of all the subschemas. This merging is a complex process called **schema integration** that will be discussed in another volume.

A pragmatic scenario to build the global conceptual schema could be as follows:

1. First, we build a project for each subsystem. Each project comprises, among others, a conceptual subschema.
2. Then, we create a new project, in which we import all the conceptual subschemas.
3. Finally, we integrate all the imported subschemas into the global conceptual schema. For this, we will use the schema integration assistant⁷ that will be described in a further volume.

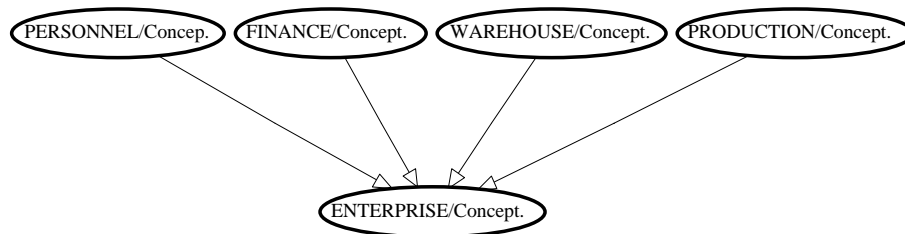


Figure 4.11 - Integration of four imported subschemas into a global conceptual schema.

This scenario is illustrated by Figure 4.11: subschemas PERSONNEL, FINANCE, WAREHOUSE and PRODUCTION have been developed independently, then have been imported into a new project. They have been integrated into the global conceptual schema ENTERPRISE.

Note about *.isl files

The ISL⁸ format is mainly intended to exchange specifications between CASE tools. Hence its use in Export and Import functions. However, it has a broader

7. Called by the command **Assist / Integration**.

8. Standing for **I**nformation **S**ystem specification **L**anguage.

scope, and can replace standard repository format in some circumstances, e.g., when transferring specifications to an older version of DB-MAIN (*.lun files are upward compatible only). The main difference with *.lun projects is that *.isl projects do not include the history of the activities, and, as a consequence, they do not record the inter-product relationships.

In particular:

- A *.isl file can be opened like any *.lun project. You just have to specify in the Open project box that you want to open a *.isl file type instead of the standard *.lun.
- When you close a project, you can save it as a *.isl file.

Key ideas of Lesson 4

1. So far, a project appears as a collection of products (or documents) together with the relations between them. In its simplest form, a project is comprised of the following products: a *conceptual schema*, a *logical schema*, a *SQL program* and some *reports*.
2. A **conceptual schema** describes the concepts of an application domain, their properties and their relationships. This description is independent of any implementation technology. It is made up of entity types, attributes, relationship types and identifiers (and of more sophisticated constructs, as we will see later on).
3. A **logical schema** is the description of data structures implemented according to the model of a DBMS. A *relational logical schema* is mainly made up of tables, columns, primary keys, foreign keys and indexes.
4. The **SQL program** is the SQL expression of the data structures of the logical schema.
5. Each **report** describes some aspects of a schema of the project.
6. Schemas from a project can be **exported** to another project. Similarly, selected components of a schema can be exported as well. This technique will be used when **integrating** several schemas into a global schema.

Summary of Lesson 4

- In this lesson, we have studied the following concepts:
 - conceptual and logical schemas
 - products, which are either schemas or text files,
 - multi-product projects

- We have also learned:
 - to create and use a multi-product project
 - to make a copy of a product: **Product / Copy product**
 - to transform a schema **Transform / Relational model**
 - to generate SQL code **File / Generate**
 - to delete an object **Edit / Delete** or Del key
 - to arrange the schema windows: **Window / Tile**
 - to export (components of) schemas: **File / Export**
 - to import schemas: **File / Import**

- We have also generated and used a new kind of file:
 - import/export specifications (* .isl)

Exercises for Lesson 4

- 4.1 Open the project `SALES1` you built as a solution to Exercise 1.2. Complete this project by building a relational logical schema, and by generating a SQL program. Examine the schemas side by side, and compare them.
Can you understand some of the rules that have been applied during the schema transformation? If you don't, never mind, we will study them in detail later on.
- 4.2 Same exercise on project `STUDENT1` of Exercise 1.3.
- 4.3 Same exercise on project `LIBRARY` (or its French version `BIBLIO`). Make sure you don't save the result inadvertently, except through a **Save as** command.
- 4.4 Import in a new project that part of the schema `Library/Conceptual` which concerns books, authors and copies.
- 4.5 Open an `*.isl` file with a text processor. Can you understand its contents (or at least its main statements)? Can you say the same of a `*.lun` file?

Lesson 5

The basics of conceptual modeling

Objective

This lesson introduces the reader to the main constructs of the DB-MAIN conceptual model. In particular, s/he will learn what are optional/mandatory attributes, atomic/compound attributes, single-valued/multivalued attributes, multiple identifiers, hybrid identifiers, N-ary relationship types, relationship types with attributes, others with identifiers, and cyclic relationship types.

Preliminary checking

In this lesson, we will use the project MANU-4 (file manu-4.lun) that has been created in Lesson 4.

5.1 Starting Lesson 5

We start DB-MAIN and we open the MANU-4 project. We only keep the conceptual schema, deleting all the other products.

To prevent from possible accidents, we change the name of this project into MANU-5 and we save it under the name MANU-5.lun.

5.2 Updating an object

We have seen in lesson 3 how to update the properties of a schema (namely its Version). This technique also applies for any object of a project:

- either double-click on the object name in its Schema window, or select the object (by clicking on its name) and press the RETURN key; either of these actions opens the object property box;
- change the concerned properties of the object;
- either validate the operation by clicking on the OK button, or discard the modifications by clicking on the Cancel button.

This works fine for schemas, entity types, relationship types, attributes and groups. The only exception is the *project* itself. To modify its properties, use the command **File / Project properties** instead.

5.3 What is a conceptual schema?

Despite its limited scope, Lesson 1 has introduced some important notions about conceptual schemas. First, it showed that such schemas are technology-independent in that they comprise abstract objects that denote application domain concepts independently of their representation through DBMS constructs. The schema of Figure 5.1 has been developed by the analysis of the facts the application domain is made up of. The way these facts will be represented in terms of tables, columns and foreign keys is irrelevant at this stage.

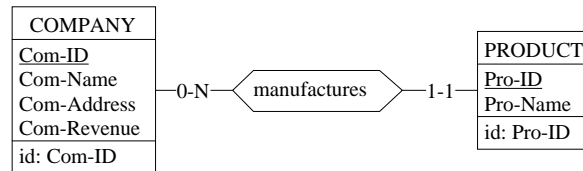


Figure 5.1 - The conceptual schema we built in Lesson 1.

This first experiment has taught us that a conceptual schema comprises entity types (COMPANY, PRODUCT), relationship types (manufactures), attributes (Com-ID, Pro-Name) and identifiers ({Com-ID}, {Pro-ID}).

An **entity type** represents a class of similar objects, or entities, that are perceived as significant when we talk about the application domain. Such objects are modeled through an entity type when we want to record information about them, when they are associated with other entities and when they obey to specific behaviour rules.

A **relationship type** (rel-type) models similar associations between the entities of two entity types. A relationship is a pair of entities, each of them belonging to one of the participating entity types. Each participating entity type plays a definite **role** in the rel-type. This role is characterized by a cardinality constraint expressed as a pair of symbols such as [1-1] or [0-N].

An **attribute** denotes a property of an entity type. It has a type (numeric, character, date, etc.), a length and a cardinality.

An **identifier** is a group of attributes that uniquely qualifies the entities of a type. At any time, two entities of this type must have distinct values for the attributes of the identifier.

In this lesson, we will discuss variants of these concepts as well as new concepts that will be useful to build more expressive conceptual schemas.

5.4 Cardinality of an attribute

Until now, we have implicitly understood that each attribute of an entity type had *one and only one* value for each entity: each COMPANY entity has one value of Com-ID, one value of Com-Name, one value of Com-Address and one value of Com-Revenue. No more, no less.

We now consider that this is not true for the latter attribute: some companies have revenues while others may have none. Therefore, some COMPANY entities have one value of Com-Revenue, while others have none. In general, we can say that any COMPANY entity has from 0 to 1 Com-Revenue value and from 1 to 1, i.e., exactly one, Com-Name value.

An interval such as 0-1 and 1-1 is called the **cardinality** of the attribute. Any couple of non-negative values is valid, provided the first one is not greater than the second one. The default value is 1-1, and is not displayed in the Schema windows.

To illustrate this concept, we double-click on Com-Revenue to open its Attribute property box, and we change its cardinality from 1-1 to 0-1, either by typing the new value or by selecting it in the listbox.

Then, we define a new attribute, named Phone-Number, that is given cardinality 1-4, stating that any company has from 1 to 4 phone numbers (Figure 5.2).

COMPANY
<u>Com-ID</u>
Com-Name
Com-Address
Com-Revenue[0-1]
Phone-Number[1-4]
id: Com-ID

COMPANY COM

Com-ID: char (15)

Com-Name: char (25)

Com-Address: char (50)

Com-Revenue: [0-1] num (12)

Phone-Number: [1-4] char (14)

id: COM-ID

Figure 5.2 - Optional and multivalued attributes.

An attribute whose cardinality has a lower bound of 0 is called **optional**. Conversely, an attribute whose cardinality has a non-zero lower bound is called **mandatory**. For instance,

- Com-Name is mandatory,
- Com-Revenue is optional,
- Phone-Number is mandatory.

An attribute whose cardinality has an upper bound greater than 1 is called **multivalued**, while those with cardinality 0-1 or 1-1, are said to be **single-valued**. For instance,

- Phone-Number is multivalued,

- Com-Name and Com-Revenue are single-valued.

5.5 Atomic and compound attributes


Some attributes can be broken down into fragments that still are significant. For instance, any value of Com-Address can be seen as composed of a value of Number + a value of Street + a value of City.

Attribute Com-Address is called **compound**. Attributes Number, Street and City are its components. Note that a component can itself be compound; it is the case for City, which consists of Zip-Code and City-Name.


An attribute that is not compound is called **atomic** (i.e. *unbreakable*). For instance, Com-Name, Com-Address.Number and Com-Address.City.City-Name are atomic attributes.

Both single-valued (Com-Address) and multivalued (Phone-Number) attributes can be compound.

Changing Com-Address from *atomic* to *compound* is made by defining its components:

- We open its Attribute property box, then click on button First att., to add its first component, Number; after that, clicking on Next Att. or pressing the Enter key opens the next Attribute property box. Stop the series by clicking on the OK or Cancel button.
- Or we select attribute Com-Address in the Schema box and execute the command **New / Attribute / First**, or, equivalently, we click on the button  in the *Standard tools bar*. We introduce the other attributes as described above.

Later on, we can insert the next attributes by selecting the insertion point, then:

- either we open its property box and enter the next attributes;
- or we execute the command **New / Attribute / Next**;
- or we click on the button  in the *Standard tools bar*.

To insert an attribute in the first position, we select the entity type or the rel-type as the insertion point, then we proceed as above.

We modify the COMPANY structure as shown in Figure 5.3.

COMPANY
<u>Com-ID</u>
Com-Name
Com-Address
Number
Street
City
Zip-Code
City-Name
Com-Revenue[0-1]
Phone-Number[1-4]
Country
Area
Local
id: Com-ID

Figure 5.3 - Compound attributes.

... or, more precisely, in the Text extended view of Figure 5.4.

COMPANY / COM [S]
Com-ID: char (15) [S]
Com-Name: char (25) [S]
Com-Address: compound (50) [S]
Number: num (5)
Street: char (20)
City: compound (25)
Zip-Code: num (7)
City-Name: char (18)
Com-Revenue[0-1]: num (12) [S]
Phone-Number[1-4]: compound (14)
Country: num (3)
Area: num (3)
Local: num (8)
id: Com-ID

Figure 5.4 - Compound attributes in the Text extended view.

Note that a compound attribute has a length too, as shown in its Property box. However, this length is calculated, and cannot be changed through the Attribute box itself.

You probably have observed that entity types, relationship types and even groups (e.g., identifiers) have been assigned a length field as well¹. Its value is the sum of the lengths of their attributes or components, if any.

5.6 Multiple identifiers

An entity type can have more than one identifier. Let us consider the entity type COMPANY. It is identified by its attribute Com-ID, which means that, in the database described by the schema, no two COMPANY entities will be allowed to share the same value of Com-ID.

In addition, we assume that there are no two companies with the same name and the same address. Therefore, we will specify a second identifier, comprising Com-Name and Com-Address: we select these attributes simultaneously (use the Shift key) and we click on the ID button in the *Standard tools bar* as shown in Lesson 1 (Figure 5.5).

COMPANY
<u>Com-ID</u>
Com-Name
Com-Address
Number
Street
City
Zip-Code
City-Name
Com-Revenue[0-1]
Phone-Number[1-4]
Country
Area
Local
id: Com-ID
id':Com-Name
Com-Address

Figure 5.5 - An entity type with two identifiers. The first one {Com-ID} is primary while the second one {Com-Name,Com-Address} is secondary.

1. The length is followed by a "+" sign when the object includes role components whose length cannot be evaluated.

If an entity type has identifier(s), one of them generally is declared **primary** (notation id), while the others, if any, are declared **secondary**, and are noted id' instead.



Note that an entity type can have secondary identifiers only. However, it can have only one primary id. It is a good practice to define the most natural identifier as primary. The problem of choosing identifiers can be a bit more complex, and will be discussed later on.

Note: An entity type need not have identifiers. An entity type without any explicit identifier is an infrequent, but quite valid situation.

5.7 Hybrid identifiers

Until now, an identifier consisted of one or several attributes of the entity type. In some situations however, an identifier can be more complex.

To illustrate this point, we need a more sophisticated schema. We suppose that a company comprises branches, and that products are manufactured by branches, not by companies. Therefore:

- We create entity type BRANCH, with attributes Name (identifier) and Country.
- We create rel-type belongs between BRANCH and COMPANY by drawing a line (use the button ) from COMPANY to BRANCH.
- In manufactures, we replace COMPANY with BRANCH as follows: we delete the role COMPANY and we draw a line (with the button ) between manufactures and BRANCH.

In addition, let us suppose that all the branches of the same company are located in distinct countries, but that two branches of different companies may be installed in the same country.

Such a situation can be described by stating that a BRANCH entity *is identified by the COMPANY it belongs to + its COUNTRY*. This identifier, made of attributes and of roles, is called **hybrid**.

A hybrid identifier is defined in the same way as *all-attribute* identifiers: we select the attribute Name and the role manufactures . COMPANY, then we click on the ID button (Figure 5.6).

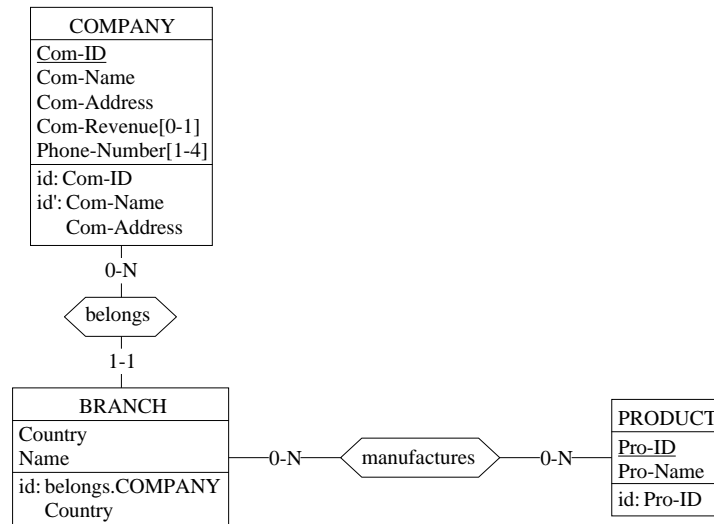


Figure 5.6 - Entity type BRANCH has a hybrid identifier comprising a remote role and a local attribute. The components of compound attributes have been hidden for simplicity.

The identifier of an entity type can be made up of one of the following combinations:

- one or more local attributes;
- two or more remote roles (hybrid);
- one or more local attributes + one or more remote roles (hybrid).

It is interesting to further analyze the position of rel-types in constructing identifiers. For instance, why have we discarded *identifiers made of one role only*?

Let us examine the examples of Figure 5.7. While a customer can place any number of orders, each order has been placed by one customer only. So, we can say that *each order identifies a customer*. Similarly, each vehicle can be used by one salesman only, so that *vehicles identify salesmen*.

We could be tempted to declare these identifiers explicitly:

- for CUSTOMER: id: places.ORDER
- for SALESMAN: id: uses.VEHICLE

However, it would be useless to declare them since they can be inferred from the cardinality of the rel-types. Indeed, ORDER is an identifier for CUSTOMER thanks to its cardinality [1-1] in `places`, and VEHICLE is an identifier for SALESMAN due to cardinality [0-1] in `uses`.

These identifiers are called **implicit**, and must not be declared. Anyway, DB-MAIN will not allow us to define them (try it to check!).

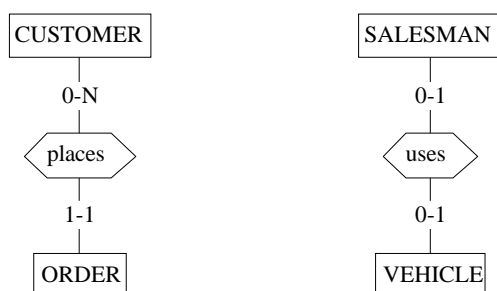


Figure 5.7 - Implicit entity type identifiers.

The concept of identifier can be richer than it appears in this lesson. It will be discussed further later on.

5.8 On defining identifiers

The way we defined identifiers is very intuitive: selecting a group of components, then telling (through the button ID) that they form an identifier.

This is just a short-hand for a more general technique that will be necessary later on. So, it is useful to describe it now.

The idea is to select the entity type, then to create a new identifier for it. To experiment with it, first delete the secondary id of COMPANY: select the group labelled 'id', then press the Del key. Now we will define it again in another way.

We select entity type COMPANY and we execute the command **New / Group**. A new property box opens. It allows us to define a **group** of attributes and/or roles that plays some outstanding functions with respect to their entity type. Once the Property box is opened, we proceed as follows:

- we specify what are the components of the group by selecting Com-Name and Com-Address in the right list and adding them to the left list thanks to the buttons Add First and Add Next;
- we tell that this group of components forms a *secondary identifier* by checking the Secondary ID button (Figure 5.8);
- we confirm this choice by clicking on button OK.

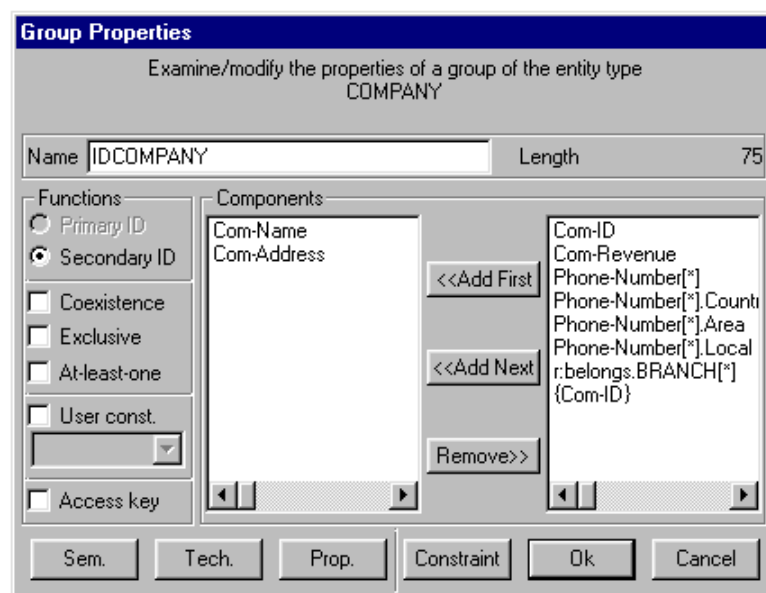


Figure 5.8 - A secondary identifier of COMPANY is being defined.


As will be discussed later on, this box will be used to define many other sophisticated constraints.

5.9 N-ary relationship types

The relationship types we have defined so far are made of two roles, and therefore are called *binary*. It is possible to define relationship types with three (or more) roles. They are called **N-ary rel-types**, where N is the number of roles, also called the **degree** of the rel-type.

In the following schema, we have defined a new entity type, namely MARKET, that represents the different markets on which products can be distributed. In addition, we have considered that a branch manufactures products for some markets only. Therefore, each manufactures relationship links one BRANCH entity (say B), one PRODUCT entity (P) and one MARKET entity (M). Such a relationship expresses the fact that:

branch B manufactures product P for market M.

We can change relationship type manufactures from binary (2 roles) to ternary (3 roles) in a simple way: we draw a line (button ) from manufactures to MARKET (Figure 5.9). We improve the layout of the schema by *right-clicking* on the rel-type.

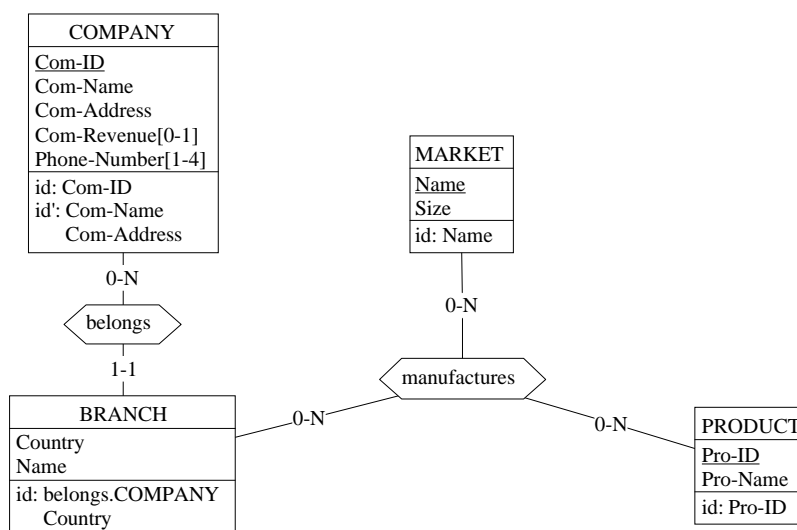


Figure 5.9 - A rel-type linking 3 entity types. Now, *branches manufacture products for markets*.

5.10 Relationship types with attributes

Attributes can be associated with relationship types as well. Let us suppose that the manufacturing of a product P by a branch B for a given market M is measured by a *ratio* that states what part of the production of product P goes to market M from branch B.

The attribute describing this ratio is created in the same way as for entity types. For instance:

- we open the property box of `manufactures`;
- we click on the New Att. button and we define the attribute.

The schema should look like in Figure 5.10.

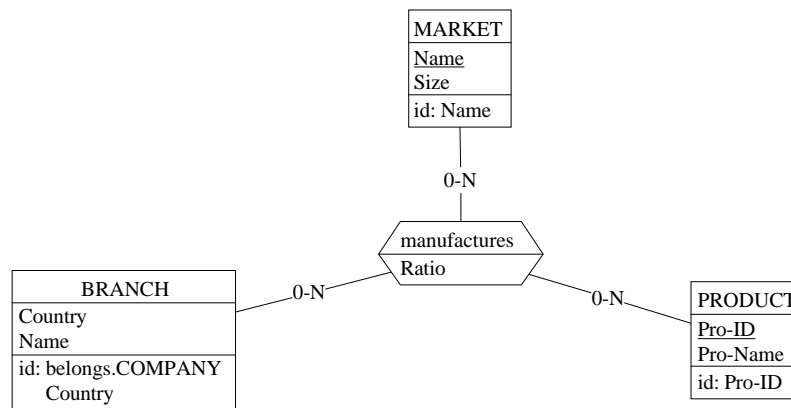


Figure 5.10 - A rel-type can be given attributes as well.

5.11 Relationship types with identifier(s)

Relationship types can have identifiers too. For instance, we could imagine a new rule of the application domain stating that,

when a branch manufactures a product, it does it for one market only.

Considering `PRODUCT` entity `P` and `BRANCH` entity `B`, the database could not include more than one `manufactures` relationship in which both `P` and `B` appear. Therefore there is at most one `MARKET` entity associated with any couple of entities $\langle P, B \rangle$. This property can be expressed by an **identifier** of `manufactures` comprising `PRODUCT` and `BRANCH`.

Such an identifier cannot be defined by simply clicking on the ID button, as for entity types, due to the ambiguities that may arise in some situations². Instead, we will use the general technique described in Section 5.8:

-
2. There is no ambiguity when the identifier comprises a local attribute. In this case, the ID button is active, and can be used as for entity types.

- we select `manufactures` by clicking on its name;
- we execute the command **New / Group** to open the Group property box;
- we select the roles `manufactures.PRODUCT` and `manufactures.BRANCH`³ and we move them in the left list;
- we check the button **Primary ID** and we confirm the operation (Figure 5.11).

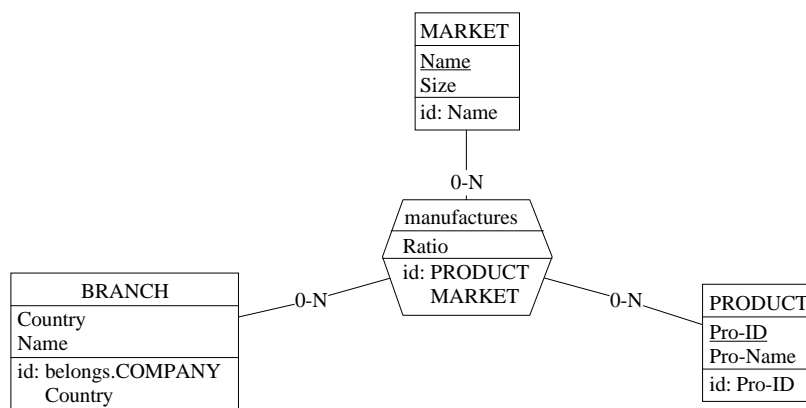


Figure 5.11 - A rel-type can have explicit identifiers.

In fact, unlike entity types, every relationship type has (at least) one identifier, but most of them should not be declared explicitly as illustrated here above. An intuitive principle could be that *there cannot exist two relationships of a given type between the same entities*. We can tell that *branch B belongs to company C*, but it is unnecessary to say it twice!

DB-MAIN will consider as an **implicit identifier** of relationship type R ,

- each role of R with cardinality 0-1 or 1-1,
- all the roles of R when R has no such 0-1 or 1-1 roles, and when no explicit identifiers have been declared.

For instance, the (implicit) identifier of relationship type `belongs` is `BRANCH`, and the (implicit) identifier of `manufactures` in `MANU-4` was `(COMPANY, PRODUCT)`. Therefore, such identifiers need not be declared, DB-MAIN being able to cope with them adequately.

3. Note that the roles are prefixed with r : in the Property box to distinguish them from attributes.

In short, an explicit identifier of a rel-type can be made up of one of the following combinations:

- two or more local roles;
- one or more local attributes + one or more local roles;
- one or more local attributes (infrequent but valid).



5.12 Cyclic relationship types

Each role of a relationship type is defined as the participation of an entity type. A relationship type in which the same entity type participates more than once is perfectly valid.

Let us consider that a product can be replaced, when unavailable, with another product, called its substitute. This fact can be represented easily by relationships between some PRODUCT entities and other PRODUCT entities. Such relationships form a **cyclic** relationship type.

To represent this, we define a new relationship type, with name `replaces`, and with two roles, both defined on PRODUCT, with cardinality 0-1 and 0-N respectively (Figure 5.12).

Here, we have a problem: *DB-MAIN does not let us draw a line between an entity type and itself!* So, we have to work in a slightly different way:

- we define the rel-type through the button  (or command **New / Rel-type**),
- then we define the roles with the button  (or command **New / Role**).

To distinguish the function of each of these roles, we will give them distinct names. The role corresponding to the product that is replaced will be called `replaced`, while the role corresponding to the product that replaces the former will be called `substitute`.

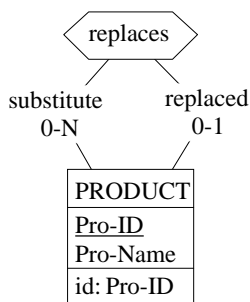


Figure 5.12 - A cyclic rel-type. Both roles have been explicitly named.

Cyclicity is not limited to binary rel-types. Indeed, rel-types of any degree can be partially or fully cyclic. Figure 5.13 shows an example of a rel-type, two roles of which have been defined on the same entity type. Its meaning is that, *at some date, a vehicle has been used to transfer a piece of equipment from a site to another one.*

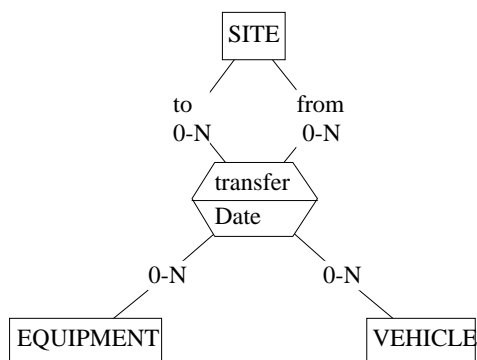


Figure 5.13 - A quaternary (4-ary) partially cyclic rel-type.

About role names

Until now, we have ignored the roles names except in cyclic rel-types. In fact, a role can be given an explicit name, be it part of a cyclic rel-type or not. In the schema of Figure 5.14, we have given roles explicit role names to stress the specific role each member plays in the rel-type. When we give a role no name, DB-MAIN gives it, as default name, that of the participating entity type.

For instance, the relationship type `belongs` in Figure 5.9 has two roles with default names `COMPANY` and `BRANCH`, though we gave them no explicit names.

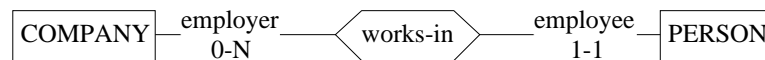


Figure 5.14 - A rel-type with explicit role names.

This being said, we can state a property each relationship type must satisfy:

the roles of a rel-type have distinct names, be they explicit or default.

Applying this property to cyclic relationship types means that all their roles (or all of them but one) must receive explicit distinct names.

Since the same role name may appear in several relationship types, its name alone cannot identify it in its schema. Therefore, the full name of a role includes also that of its relationship type. For instance, the roles of `belongs` have full names `belongs.BRANCH` and `belongs.COMPANY`, and those of `replaces` have names `replaces.replaced` and `replaces.substitute`. Accordingly, these full names appear in the listboxes of the Group property boxes and in the specification of the groups in the schemas.

Cyclic, unary, recursive and reflexive rel-types

It must be noted that the term *cyclic* is not standardized, and that other names will be used in the literature.

Some authors consider that degree N is not the number of roles, but rather the number of distinct participating entity types. Hence the concept of **unary** rel-type, that will be called in this model *cyclic binary rel-type* instead, to comply with the mathematical definition of relations.

Other authors call cyclic rel-types **recursive**. This term rather qualifies algorithms that use such rel-types as well as other structures which includes circuits of rel-types.

The term **reflexive** is also sometimes used to designate cyclic rel-types. We will avoid this term, since it has a well defined mathematical definition that⁴ does not stand in arbitrary cyclic rel-types.

Finally, the term acyclic itself can be disputed. Indeed, the qualifier *acyclic* is used to designate a (directed) relation in which no cycles are allowed (such as *parent*, defined from *persons* to *persons*). Therefore a *cyclic rel-type* could be misleadingly interpreted as defined by a relation in which cycles are allowed.

5.13 The complete schema

If all the extensions described above have been included, the schema should appear as in Figure 5.15 or Figure 5.16.

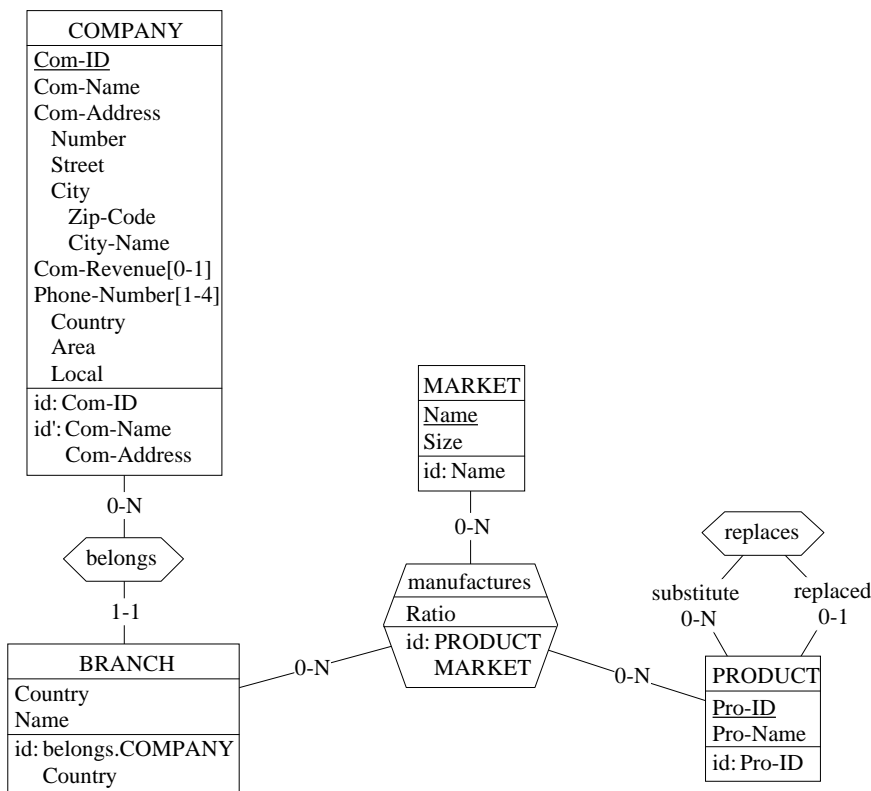


Figure 5.15 - The Graphical standard view of the final schema.

-
- A relation $R(A,A)$ is reflexive if, for any element a of A , $\langle a,a \rangle \in R$.

```

Schema Manufacturing/Conceptual-Final

BRANCH
  Country
  Name
  id: belongs.COMPANY, Country

PRODUCT
  Pro-ID
  Pro-Name
  id: Pro-ID

COMPANY
  Com-ID
  Com-Name
  Com-Address
    Number
    Street
    City
    Zip-Code
    City-Name
  Com-Revenue[0-1]
  Phone-Number[1-4]
    Country
    Area
    Local
  id: Com-ID
  id': Com-Name, Com-Address

MARKET
  Name
  Size
  id: Name

manufactures (
  [0-N]: BRANCH
  [0-N]: PRODUCT
  [0-N]: MARKET
  Ratio )
  id: PRODUCT, MARKET

belongs (
  [0-N]: COMPANY
  [1-1]: BRANCH )

replaces (
  substitute [0-N]: PRODUCT
  replaced [0-1]: PRODUCT )

```

Figure 5.16 - The Text standard view of the final schema.

5.14 On the cardinalities of rel-types

Let us first recall the meaning of the cardinality of a role. Considering rel-type RT with roles *ra* and *rb* defined as follows (Text standard):

```
RT(
  ra[ia-ja]: A
  rb[ib-jb]: B)
```

role "*ra*[*ia-ja*]: A" states that

any A entity appears in role ra in at least ia and in at most ja RT relationships.

The way of understanding the concept cardinality can be called *the participation interpretation*, because it measures the number of *participations* of each entity. According to it, cardinality [*ia-ja*] is a constraint on entity type A⁵.

Binary rel-types

It is common practice to give some configurations specific names as follows:

R is called ...	if ...
<i>one-to-one</i>	$ja = jb = 1$
<i>one-to-many</i> from A to B	$ja > 1$ and $jb = 1$
<i>many-to-one</i> from A to B	$ja = 1$ and $jb > 1$
<i>many-to-many</i>	$ja > 1$ and $jb > 1$

In addition role *ra* will be called:

ra is called ...	if ...
<i>optional</i> for A	$ia = 0$
<i>mandatory</i> for A	$ia > 0$

5. Some models such as OMT and UML use another interpretation of cardinalities. Both will be discussed in another lesson.

The schema of Figure 5.17 shows some examples of this classification:

- owns is *one-to-many* for CUSTOMER ,
- owns is *many-to-one* for VEHICLE ,
- involved is *many-to-many*,
- covered by is *one-to-one*,
- covered by is *mandatory* for VEHICLE,
- covered by is *optional* for INSURANCE.

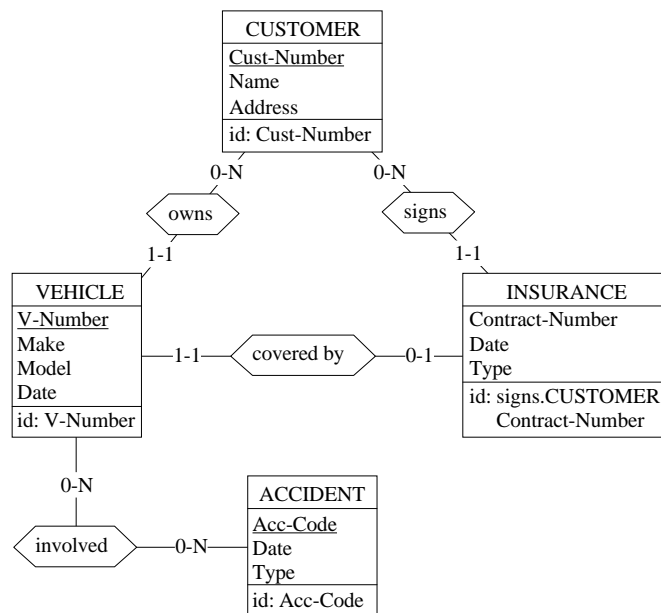


Figure 5.17 - A *potpourri* of rel-types.

Inasmuch as one-to-many, many-to-one and one-to-one rel-types materialize *mathematical functions*, they also are called **functional** rel-types.

As a last definition, we will call **binary schema** any schema in which all rel-types are *binary* and have *no attributes*.

N-ary rel-types

The classification from *one-to-one* to *many-to-many* generally is not applicable when $N > 2$. However, some authors generalize it by using terms such as *many-to-many-to-many* (e.g., `manufactures` in Figure 5.9) or *one-to-many-to-many*.

Hence the term *many role* to designate a role with $j > 1$ and *one role* to designate a role with $j = 1$. A one-to-many rel-type has 1 *many role* and 1 *one role*.

5.15 Minimal identifiers

Let us go back to the schema of Figure 5.9. It tells us that there cannot exist two companies sharing the same `Com-ID` value. But what about two companies with the same values of `Com-ID` and `Com-Name`? Of course, there cannot be more than one either.

The same reasoning can be held for any value set of combination `{belongs.COMPANY, Name, Country}`, which obviously designates at most one `BRANCH` entity.

We can conclude that,

- `{Com-ID, Com-Name}` is an identifier of `COMPANY`,
- `{belongs.COMPANY, Name, Country}` is an identifier of `BRANCH`.

Of course, we feel that these are not *good* identifiers. We shall say that they are not minimal. A **minimal identifier** is a group of attributes and/or roles such that any strict subset is no longer an identifier.

Needless to say, we will avoid declaring non-minimal identifiers, and that we can keep the schema of Figure 5.9 as is. Practically speaking,

- we will discard any identifier a subset of which is an implicit or declared identifier⁶,
- we will carefully examine each multi-component identifier to make sure that none of its components can be discarded.

6. As we will see later on, the Schema Analysis assistant of DB-MAIN can detect these situations.

5.16 What next?

There are many other useful conceptual constructs to be discussed. However, those we described in this lesson are quite sufficient to build complex databases. Therefore, before learning about advanced conceptual structures, we will get a deeper insight into logical structures such as those we caught a glimpse of in Lesson 4.

Key ideas of Lesson 5

1. Entity types

An *entity type* represents a class of concrete or abstract real-world entities, such as customers, orders, books, cars and accidents.

An entity type can comprise attributes, can play roles in rel-types, and can be given identifiers.

2. Relationship types (rel-types)

A *relationship type* represents a class of associations between entities. It consists of entity types, each playing a specific role. A rel-type with 2 roles is called *binary*, while a rel-type with more than 2 roles is called *N-ary*. A rel-type with at least 2 roles taken by the same entity type is called *cyclic*.

Each role is characterized by its *cardinality* $[i-j]$, a constraint stating that any entity of this type must appear, in this role, in i to j associations or relationships. Generally i is 0 or 1, while j is 1 or N (= *many* or *infinity*). However, any pair of integers can be used, provided that $i \leq j$, $0 \leq i$ and $0 < j$.

A *binary rel-type* between A and B with cardinality $[i_a-j_a]$ for A, $[i_b-j_b]$ for B is called:

- one-to-one if $j_a = j_b = 1$
- one-to-many from A to B if $j_a > 1$ and $j_b = 1$
- many-to-one from A to B if $j_a = 1$ and $j_b > 1$
- many-to-many if $j_a > 1$ and $j_b > 1$
- optional for A if $i_a = 0$
- mandatory for A if $i_a > 0$.

A *one* role has cardinality $[i-1]$, while a *many* role has cardinality $[i-j]$ with $j > 1$. A binary rel-type with at least one *one* role is called *functional*. A *binary schema* includes only binary rel-types without attributes.

A *role* can be given a name. When no explicit name is assigned, an implicit default name is assumed, namely the name of the participating entity type. The roles of a rel-type have distinct names, be they explicit or implicit. For instance, in a cyclic rel-type, at least one role must have an explicit name.

A rel-type can have attributes, and can be given identifiers.

This model follows the *participation interpretation* of cardinalities. According to it, the cardinality of a role measures the number of relationships in which any entity appears in this role.

3. Attributes

An *attribute* represents a common property of all the entities (or relationships) of a given type. Simple attributes have a value domain defined by a data type (number, character, boolean, date, ...) and a length (1, 2, ..., 200, ..., N [standing for infinity]). These attributes are called *atomic*.

An attribute can also consist of other component attributes, in which case it is called *compound*. The *parent* of an attribute is the entity type, the relationship type or the compound attribute to which it is directly attached. An attribute whose parent is an entity type or a rel-type is said to be at level 1. The components of a level-*i* attribute are said to be at level *i*+1.

Each attribute is characterized by its cardinality [i-j], a constraint stating that each parent has from *i* to *j* values of this attribute. Generally *i* is 0 or 1, while *j* is from 1 to N (= infinity). However, any pair of integers can be used, provided that $i \leq j$, $0 \leq i$ and $0 < j$. The default cardinality is [1-1], and is not represented graphically. An attribute with cardinality [i-j] is called:

- single-valued if $j = 1$
- multivalued if $j > 1$
- optional if $i = 0$
- mandatory if $i > 0$.

4. Groups


A *group* is made of components, which are attributes, roles and/or other groups. A group represents a construct attached to a parent object, i.e. to an entity type or a rel-type. It is used to represent, among others, *identifiers*:





- *primary identifier*: the components of the group make up the main identifier of the parent object; it appears with symbol *id*; if it comprises attributes only, the latter are underlined in the graphical view; a parent object can have at most one primary id; all its components are mandatory.
- *secondary identifier*: the components of the group make up a secondary identifier of the parent object; it appears with symbol *id'*; a parent object can have any number of secondary id.

A *minimal identifier* is a group such that there is no strict subset that still is an identifier.

Summary of Lesson 5

- In this first lesson, we have studied the following concepts:
 - the cardinality of an attribute
 - single-valued / multivalued attributes
 - mandatory / optional attributes
 - atomic / compound attributes
 - multiple identifiers
 - hybrid identifiers
 - implicit identifiers
 - identifiers as a special kind of group
 - binary and N-ary relationship types
 - attributes of relationship types
 - identifiers of relationship types
 - cyclic relationship types
 - role names
 - one-to-many, many-to-one, one-to-one, many-to-many rel-types
 - functional rel-types
 - minimal identifiers.

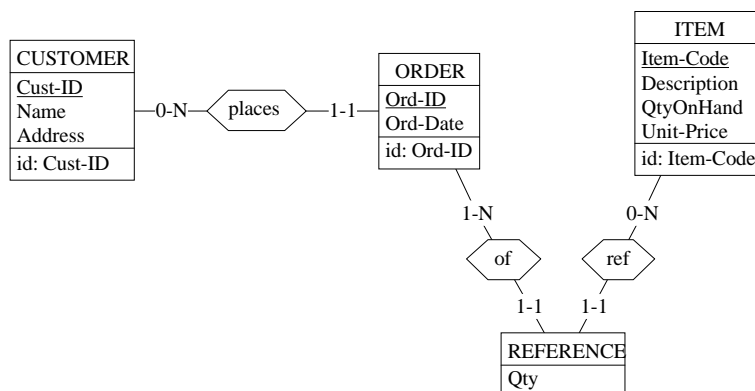
- We have also learned:
 - to update the properties of an object
 - double-click on the object description
 - File / Project properties**
 - to define the cardinality of an attribute
 - to define a compound attribute
 - New / Attribute / First** 
 - from the Attribute box* , button First att.

- to define a rel-type
New / Rel-type; New / Role  and/or 
- to add a role to a relationship type
New / Role 
- to add attributes to a relationship type
same as for entity type attributes
- to define an identifier for a relationship type
New / Group or, if local attribute: 
- to give a role a name.

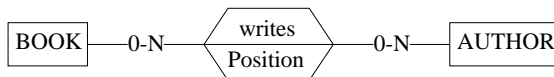
Exercises for Lesson 5

- 5.1 Build a schema describing a population of persons who have each a person id, a name, 1 to 3 christian names, possibly a maiden name, and an arbitrary number of addresses.
- 5.2 These persons may have children, who are persons too.
- 5.3 They can be married. They can even have been married several times, but only once at any given time.
- 5.4 Consider the following schema. Complete it to take into account the fact that,

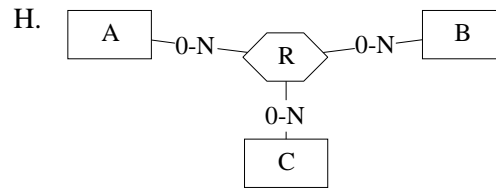
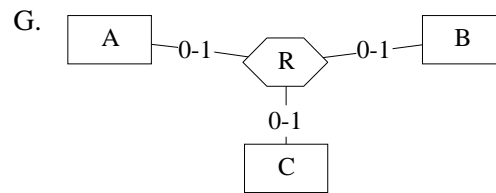
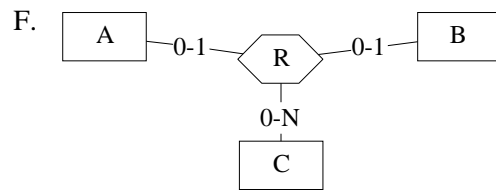
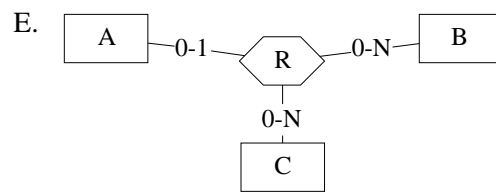
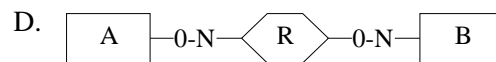
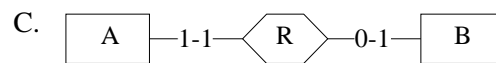
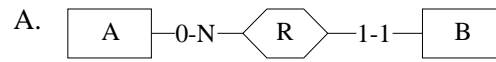
an order cannot reference an item more than once.



- 5.5 The following schema tells that *authors write books*, and that each author appears in a given position in the author list for each of his/her book (1st author, 2nd author, etc.). Complete the schema to express the following facts:
 - *an author cannot appear more than once in a book;*
 - *the authors of a book appear in distinct positions.*



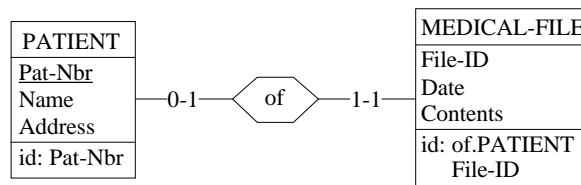
5.6 What are the **implicit identifiers** of the following rel-types?



- 5.7 Build a schema that represents customers, products and suppliers (with some natural properties such as name, address, quantity on hand, etc). Represent the fact that suppliers supply products to customers, and that they do so in a given supplied quantity and at a given date. Think very carefully about the fact that,

customer C can be supplied product P by supplier S more than once, but at different dates.

- 5.8 What do you think of the following schema?



Lesson 6

The basics of logical and physical modeling

Objective


The 6th lesson discusses some concepts of the DB-MAIN model dedicated to the representation of logical and physical constructs, i.e., components that appear in DBMS schemas as opposed to those that make up computer-independent conceptual schemas. We will describe and manipulate additional integrity constraints (e.g., referential constraints), access keys (representing indexes for instance) and entity collections (representing record files). We will also examine how to make the names of a schema comply with the syntactic rules of DBMS languages.

Preliminary checking

In this lesson, we will use project MANU-5 (file manu-5.lun) that has been created in Lesson 5.

6.1 Introduction

We start DB-MAIN and we open project MANU-5. We will work on this project, so we will first make a copy we call MANU-6:

- through **File / Project properties** we change its name into MANU-6,
- we save this version (**File / Save project as** or button ) under the name manu-6.lun.

6.2 What is a logical schema?

Lesson 4 explained how a conceptual schema can be translated into a relational schema. Both schema represent the same information, but the latter expresses it through the constructs of a DBMS¹, while the former is claimed to be DBMS-independent. A relational schema is considered to be *logical*. The same conceptual schema can be transformed into several relational logical schemas, according to the design criteria we have in mind: readability, simplicity, ease of evolution, response time, space occupied on disk, etc. In addition, considering other target DBMSs will lead to, for example, *object-relational*, *object-oriented*, *standard file*, *IMS* or *CODASYL* logical schemas.

To keep things simple, we will mainly concentrate on relational schemas, i.e., on logical schemas that comply with the relational model. Other model will be discussed in further lessons.

A relational logical schema comprises tables made up of columns, primary (or candidate) keys and foreign key. Figure 6.1 shows the logical schema we built in Lesson 4. It includes three tables, eight columns, three primary keys (id), two foreign keys (ref). In addition, it includes indexes (acc, for access keys), which have been defined on each key.

1. In other words, the conceptual structures are expressed into the model of a DBMS, or, more precisely, into the model of a family of DBMSs.

In this lesson, we will discuss in greater detail the concepts of which all relational logical schemas are made up.

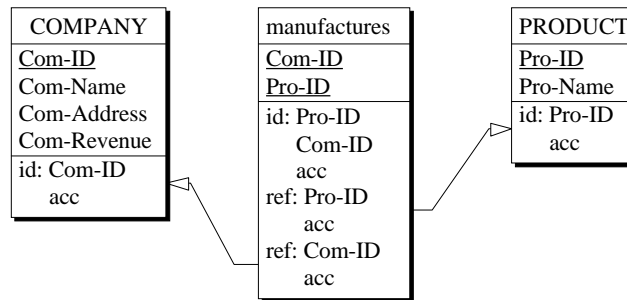


Figure 6.1 - The logical schema built in Lesson 4.

6.3 Transformation into a logical schema

Let us produce a relational logical schema for the conceptual schema we developed in Lesson 5. We proceed as suggested in Lesson 4:

- we make a copy of the schema (we select schema Manufacturing/Conceptual then execute **Product / Copy schema**) and we change its version value to "Logical";
- in this new schema, we execute **Transform / Relational model** to produce the relational structures;
- we change the graphical representation by adding shade to the entity types (**View / Graphical settings**), to make them look like tables² (with a little imagination!).

Schema Manufacturing/Logical is transformed into *relational data structures* (Figure 6.3 and Figure 6.4).

From now on, we should use the terms *table* instead of entity type, *column* instead of attribute, etc. However, the logical model is independent of specific technologies, and in particular of relational DBMS. Figure 6.2 gives the translation rules for RDBMS. Similar tables can be built for other data management systems. We will keep using the standard terms of *entity types* and *attributes*, except when mentioned otherwise.

2. The idea is that shading gives the objects a 3D look, which makes them more *concrete*.

DB-MAIN concepts	Relational terms (SQL)
entity type	table
attribute	column
primary identifier	primary key
secondary identifier	candidate key (<i>not pure SQL</i>)
reference group	foreign key
access key	index
entity collection	(table-/db-)space (<i>not standard</i>)

Figure 6.2 - Translation table of DB-MAIN names into relational names.

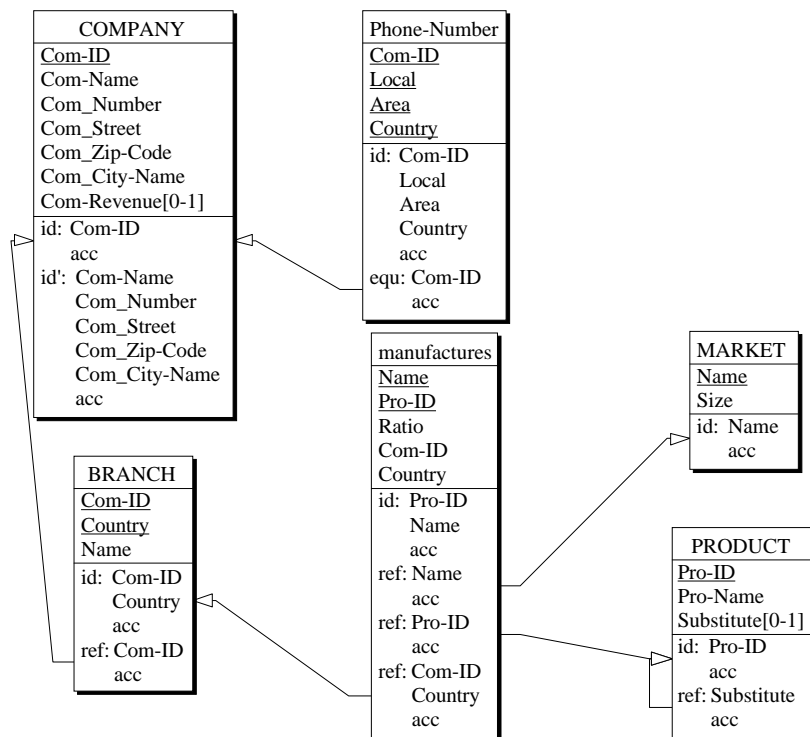
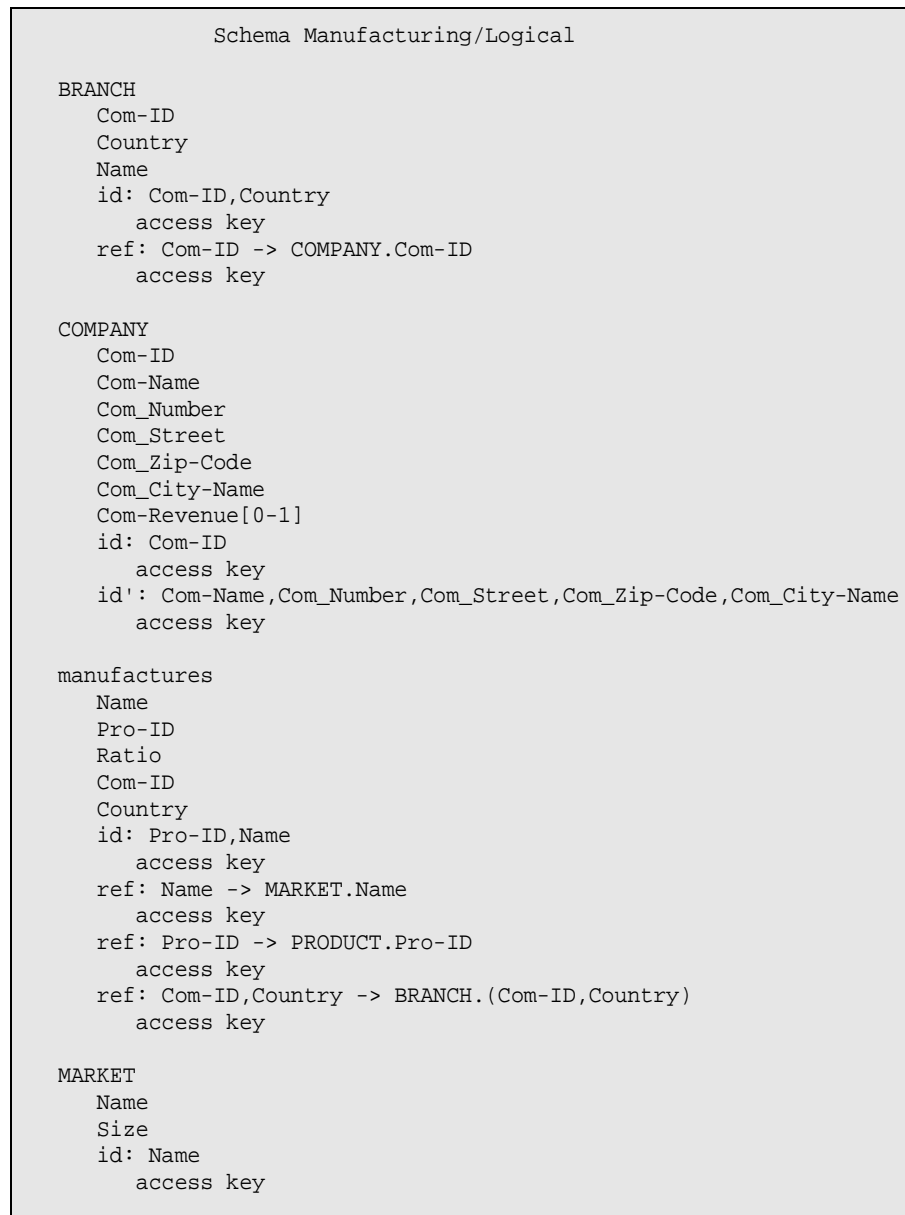


Figure 6.3 - First version of the logical schema.



```

Phone-Number
  Com-ID
  Local
  Area
  Country
  id: Com-ID,Local,Area,Country
    access key
  equ: Com-ID = COMPANY.Com-ID
    access key

PRODUCT
  Pro-ID
  Pro-Name
  Substitute[0-1]
  id: Pro-ID
    access key
  ref: Substitute -> PRODUCT.Pro-ID
    access key

```

Figure 6.4 - First version of the logical schema - Text standard view.

This schema is inevitably more complicated and less readable than its conceptual counterpart (otherwise it would have been preferable to reason from the beginning in the relational model!). Though the objective of this lesson is not to describe in detail how and why the transformation was carried out, we will try to understand, at least intuitively, the main translation rules that have been used (Section 6.6). Now, we will discuss in greater detail some important constructs that we already encountered in lesson 4, and that appear again in this schema, namely the *reference attributes* and the *access keys*.

6.4 Reference attributes (foreign keys)

A **reference attribute** is an attribute whose values act as references to other entities. For instance, attribute Com-ID in entity type BRANCH is aimed at designating a COMPANY entity. Since each entity type represents a table in this logical SQL schema, Com-ID is what is called a **foreign key** in the RDBMS language. In general, since a foreign key can comprise more than one attribute, we will talk about **reference groups**.

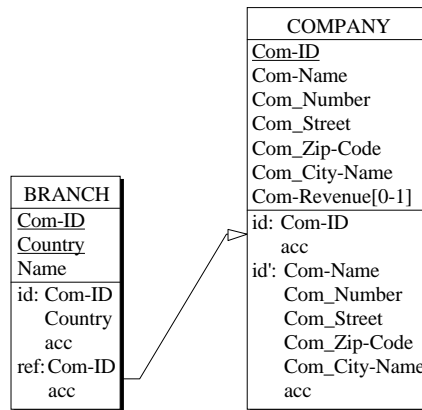


Figure 6.5 - Reference group, aka foreign key.

The way this attribute is denoted in DB-MAIN views expresses that each value of Com-ID in any BRANCH entity must be a Com-ID value in some COMPANY entity. We observe that the attribute mentioned in the target entity type (here COMPANY) is its primary identifier. In some situations, the target attribute can be a secondary identifier as well.

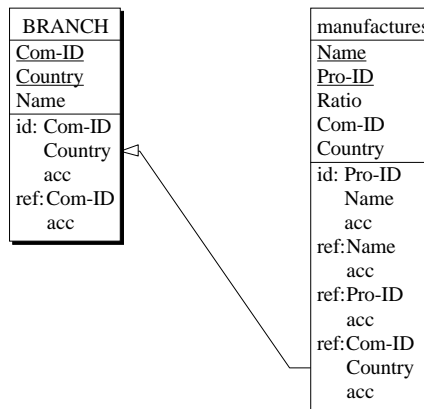


Figure 6.6 - Multicomponent reference group.

If the identifier of the target entity type is made of several attributes, then the reference must be supported by several reference attributes, as in `manufactures` entity type, where the values of attributes (`Com-ID`, `Country`) designate a `BRANCH` entity (Figure 6.6).

There is a more sophisticated form of reference attributes that can be found in entity type (i.e. table) `Phone-Number`. Let us first observe that each entity of this type represents a phone number of a company, and that all the phone numbers of company X are represented by the `Phone-Number` entities with `Com-ID = X`. Therefore, `Com-ID` is a reference attribute (or foreign key) to `COMPANY`.

However, the conceptual schema tells us that each company must have **at least one** phone number (cardinality [1-4]). This property translates, in the current logical schema, into a constraint stating that each `COMPANY` entity must have at least one corresponding `Phone-Number` entity. More precisely, the value of `Com-ID` of each `COMPANY` entity must match the `Com-ID` value of at least one `Phone-Number` entity.

Since the `COMPANY.Com-ID` values form a subset of the `PHONE-NUMBER.Com-ID` values and the `PHONE-NUMBER.Com-ID` values form a subset of the `COMPANY.Com-ID` values, we can conclude that,

the set of `COMPANY.Com-ID` values is equal to the set of `PHONE-NUMBER.Com-ID` values.

To represent this constraint, `DB-MAIN` uses the term `equ`, that expresses that the value sets of `Com-ID` in both entity types are *equal* (Figure 6.7).

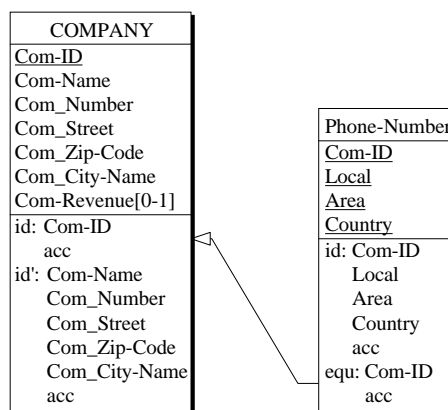



Figure 6.7 - Equality reference group.

So far, referential attributes are automatically defined as the representation of relationship types. Later on we could find it useful to define referential constraints manually, for instance to document an existing SQL database.

To practice defining referential attributes, we delete the last constraint of entity type `manufactures` by clicking on the "`ref: Com-ID, Country`" line, and pressing the Del key. The line disappears.

To build it again, we define for entity type (table) `manufactures`, a group of attributes comprising `COM-ID` and `Country`:

- we select both attributes, and we click on the button  in the *Standard tools bar* (Figure 6.8);
- we open the Property box of this group (press the Enter key or double-click) (Figure 6.9).

manufactures
<u>Name</u>
<u>Pro-ID</u>
Ratio
Com-ID
Country
id: Pro-ID
Name
acc
ref:Name
acc
ref:Pro-ID
acc
gr: Com-ID
Country

Figure 6.8 - A group comprising {`Com-ID`, `Country`} has been defined

Now we have to tell DB-MAIN that this group is a reference to table `BRANCH`. We click on the Constraint button (for *inter-group constraint*). The Constraint box opens. We have two properties to specify:

- what *kind of constraint* do we want? Let us click on the Ref button;
- what is the *target entity type*, and what is the *target identifier*? DB-MAIN will help us considerably by suggesting candidate entity types, and for each of them suggesting candidate identifiers. These suggestions are based on

the structure of the source group we have built, i.e., its composition, the type and the length of its components. In this case, there is not much choice: only the BRANCH entity type has an identifier composed of two attributes whose types and lengths match those of the current group. Therefore, DB-MAIN proposes this target entity type and this identifier only.

To make this schema equivalent to its former version, don't forget to click on the Access key button as well (more on this below).

DB-MAIN includes specific tools for finding foreign keys, such as DDL extractors and the Foreign key Assistant. They will be studied in other lessons devoted to Reverse Engineering.

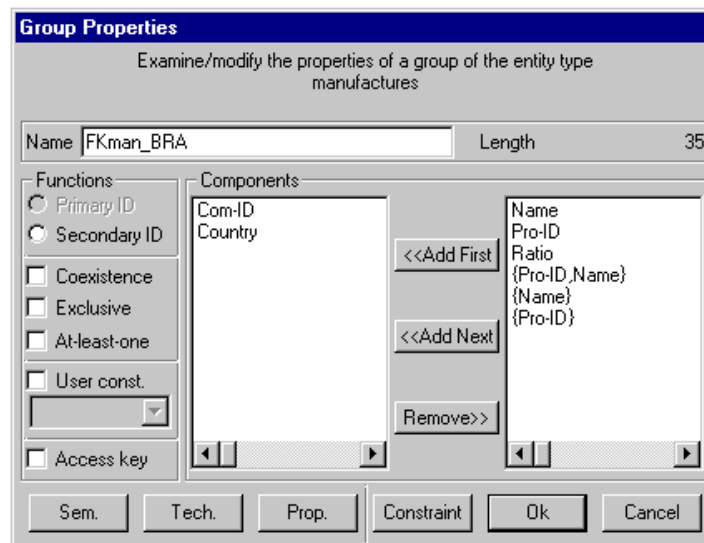


Figure 6.9 - The properties of the newly defined group.

Note

In this section, we have described foreign keys made of single-valued attributes. In Lesson 7, we will consider more complex forms of foreign keys that can be found in non-relational³ or in post-relational⁴ databases.

3. Such as in COBOL files.

4. Such as in Object-oriented and SQL-3 databases.

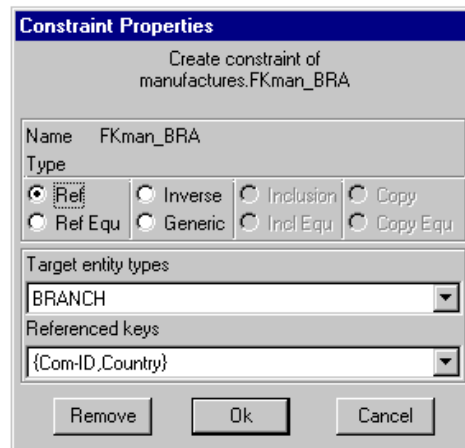


Figure 6.10 - Choosing the target of the reference group.

6.5 Access keys

The transformation has generated *access keys*. This term designates technical data structures that provide efficient selective access to data records. An access key will generally be implemented as an *index* or a *hash table* in relational DBMS. However, the term *access key* has been chosen instead of *index* since each DBMS generally proposes its own names to denote these techniques⁵.

Let us consider entity type MARKET (Figure 6.11). Its attribute Name is declared both **identifier** (id) and **access key** (acc or access key). Indeed, RDBMS generally require that each identifier be an index as well. This means that Name is an identifier, and, in addition, an access key. Therefore, asking for the MARKET whose Name is known will lead to a quick answer from the database.

5. Let us cite *record keys* in COBOL files and *calculated keys* in CODASYL databases.

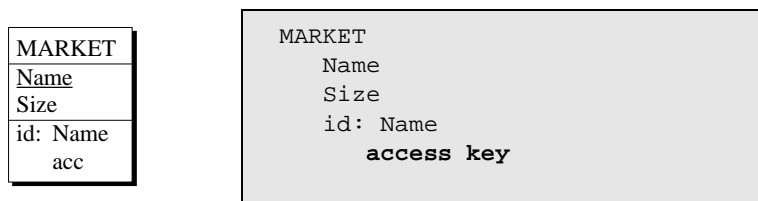


Figure 6.11 - Identifiers often are supported by access keys.

In addition, all the reference groups (foreign keys) have been made access keys as well (Figure 6.12). It is not mandatory, but DB-MAIN has found it handy to propose this in its transformation process. Indeed, such attributes implement relationship types, and therefore will most probably be used as selection criteria in the programs (in join-based queries for instance).

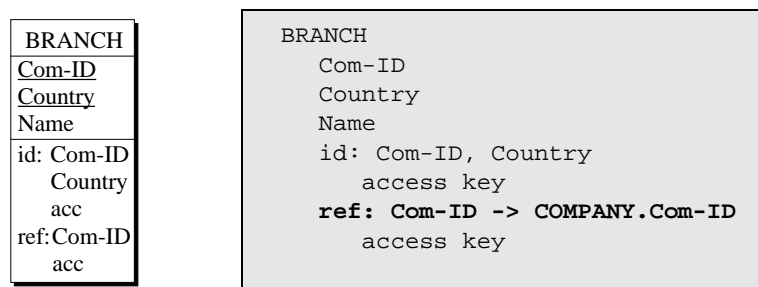



Figure 6.12 - Reference groups (foreign keys) are supported by access keys.

So far, an access key is just an additional property of another construct (identifier or referential group). We can also decide to declare other access keys if we think they can boost the performance of queries.

For instance, we can consider that asking for a product of which only the name is known, is a frequent query. To accelerate the processing of this query, we decide to build an access key on `Pro-Name` of `PRODUCT`.

An access key is just a special kind of a *group*. To illustrate it, we add a new group to `PRODUCT`:

- we select attribute the `Pro-Name`, and we click on the button  ;

- we open the Property box of this group (press the Enter key or double-click) (Figure 6.9);
 - we click on the button `Access key` and we confirm the operation.
- The entity type `PRODUCT` now reads as in Figure 6.13.

PRODUCT	
<u>Pro-ID</u>	
Pro-Name	
Substitute[0-1]	
id: Pro-ID	
acc	
ref: Substitute	
acc	
acc: Pro-Name	

PRODUCT	
Pro-ID	
Pro-Name	
Substitute [0-1]	
id: Pro-ID	
access key	
ref: Substitute -> PRODUCT.Pro-ID	
access key	
access key: Pro-Name	

Figure 6.13 - An additional access key.

6.6 On the conceptual → relational translation rules

By comparing the conceptual schema with the logical schema of Figure 6.3, we can guess the main translation rules that have been used to produce the latter schema from the former one.

First, it is clear that each **entity type** is represented by a table, and that each atomic, single-valued attribute is represented by a column (Figure 6.14).

A **compound attribute** is translated in as many columns as it has atomic components, and this, recursively (Figure 6.15).

A **multivalued attribute** is represented by a new table, that includes the column(s) of the attribute + a foreign key that references the source entity type (Figure 6.16).

<table border="1" style="margin: auto;"> <tr><th colspan="2">MARKET</th></tr> <tr><td><u>Name</u></td><td></td></tr> <tr><td>Size</td><td></td></tr> <tr><td>id: Name</td><td></td></tr> </table>	MARKET		<u>Name</u>		Size		id: Name		<table border="1" style="margin: auto;"> <tr><th colspan="2">MARKET</th></tr> <tr><td><u>Name</u></td><td></td></tr> <tr><td>Size</td><td></td></tr> <tr><td>id: Name</td><td></td></tr> <tr><td>acc</td><td></td></tr> </table>	MARKET		<u>Name</u>		Size		id: Name		acc	
MARKET																			
<u>Name</u>																			
Size																			
id: Name																			
MARKET																			
<u>Name</u>																			
Size																			
id: Name																			
acc																			

Figure 6.14 - Translation of a conceptual entity type into a table.

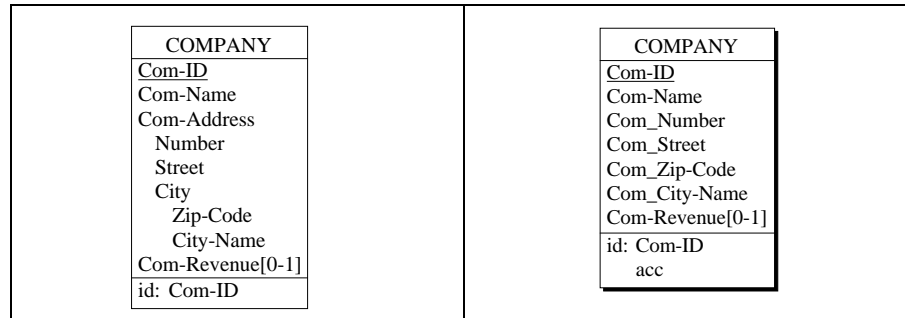


Figure 6.15 - Translation of compound attributes into a series of columns.

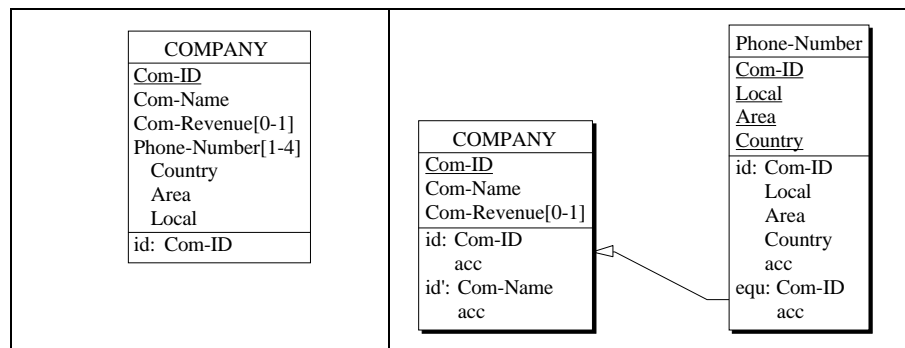


Figure 6.16 - Translation of a multivalued attribute into a secondary table.

There are two kinds of rel-types, namely *functional* rel-types (one-to-one or one-to-many) and *non-functional* rel-types (many-to-many or N-ary or with attributes), also called *complex*.

A **functional rel-type** is represented by a foreign key from the *one* side to the *many* side, if any (Figure 6.17). In case of a cyclic rel-type, the foreign key references its own table.

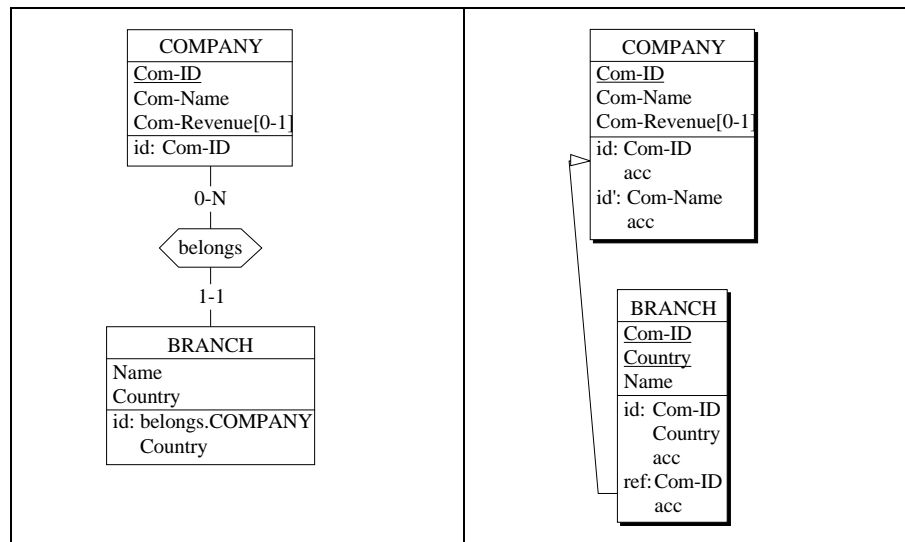


Figure 6.17 - Translation of a functional rel-type into a foreign key.

A **complex rel-type** is translated into a new table, which materializes the relationships, hence the name *relationship table*. Each role of the rel-type becomes a foreign key of the relationship table toward the table of the entity type playing this role. The rel-type attributes translate into columns of the relationship table. Identifiers are made explicit (Figure 6.18).

As discussed in Section 6.5, each identifier and each foreign key has been made an access key. Due to the lack of information on performance requirements, there is no possibility to generate other access keys, and none have been defined!

We can guess that these are not the only possible translation rules, and things can get more complex for larger and more sophisticated schemas. In addition, taking into account additional requirements such as response time, update time or disk space reduction can require more refined translation rules. They will be addressed later on⁶.

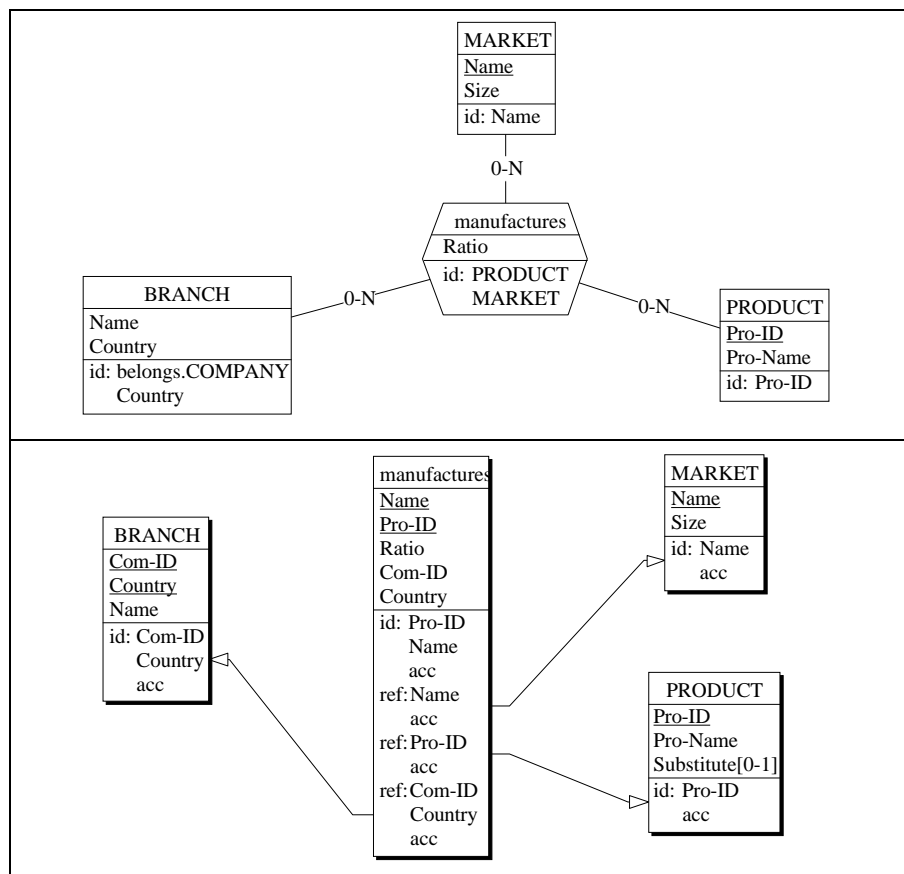


Figure 6.18 - Translation of a complex rel-type into a relationship table.

-
6. We want to mention just a simple rule that could improve the schema obtained so far: under certain conditions, when an index has been defined on columns $\langle A, B, C \rangle$, any index defined on a *prefix* subset such as $\langle A, B \rangle$ or $\langle A \rangle$ can be discarded without performance penalty. In this way, 3 indexes can be removed from the logical schema. This rule will be discussed in Volume 2 of this tutorial.

6.7 Defining entity collections

In a real database, that is, one which is implemented in an actual computer, table rows and records are stored in a large secondary memory, such as on a magnetic disk. More specifically, they are stored in storage units called, depending on the data management system, *files*, *data files*, *datasets*, *areas*, *realms*, *DBspaces* or *tablespaces*.

DB-MAIN proposes a concept to represent such storage units, namely the **entity collection**, or, more simply, the **collection**.

Let us suppose that the six tables of the relational database have to be stored into two distinct files, one, called PR_STORE, which can accommodate the rows of PRODUCT, MARKET and manufactures, and the other, called CY_STORE, in which the rows of COMPANY, BRANCH and Phone-Number will be stored.

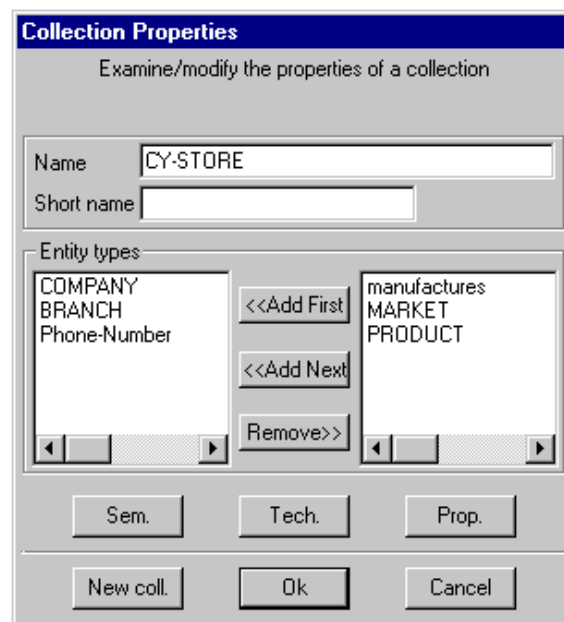



Figure 6.19 - Defining the entity collection CY_STORE.

A collection is created through the button  and specified through the *Collection property box*, called up by pressing the Enter key or by double-clicking

on the name of the collection (Figure 6.19). It allows us to specify the name, short name, semantic and technical (see below) descriptions, and the list of the entity types (or *tables*) whose entities (*rows*) are to be stored in the collection. Any number of entity types can be stored in a collection, and an entity type can be *stored* in any number of collections. However, some DBMS can impose more limited configuration. For instance, many relational DBMS force the rows of each table to be stored in one table space only, though the latter can receive rows from several tables.

These collections appear in all schema views (Figure 6.20 and Figure 6.21).

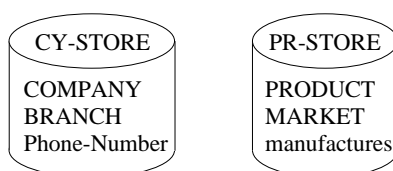


Figure 6.20 - Entity collections: storage units to store table rows in.

6.8 Name processing

Now we could believe that we are ready to generate the SQL schema that corresponds to the final version of our relational database.

However, a quick look at this schema will show a little but potentially annoying problem: some names include the character "-" (dash), which is invalid in SQL data names. A standard remedy consists in replacing each character "-" by, say, the character "_" (underscore). For instance, Com-ID should be replaced by Com_ID, and so on.

```

Schema Manufacturing/Logical / Manu

collection CY-STORE
  COMPANY
  BRANCH
  Phone-Number
collection PR-STORE
  PRODUCT
  MARKET
  manufactures

BRANCH
  in CY-STORE
  Com-ID: char (15) [S]
  Name: char (1)
  ...$

COMPANY / COM [S]
  in CY-STORE
  Com-ID: char (15) [S]
  Com-Name: char (25) [S]
  ...

manufactures [S]
  in PR-STORE
  Name: char (24)
  Pro-ID: char (8) [S]
  ...

```

Figure 6.21 - Entity collections, according to the Text extended view.

DB-MAIN has a specific processor for that task. It is called up through **Transform / Name processing**, which opens the *Name Processing panel* (Figure 6.22). We will examine this processor in detail in a further lesson, but we can already use it to solve our problem.

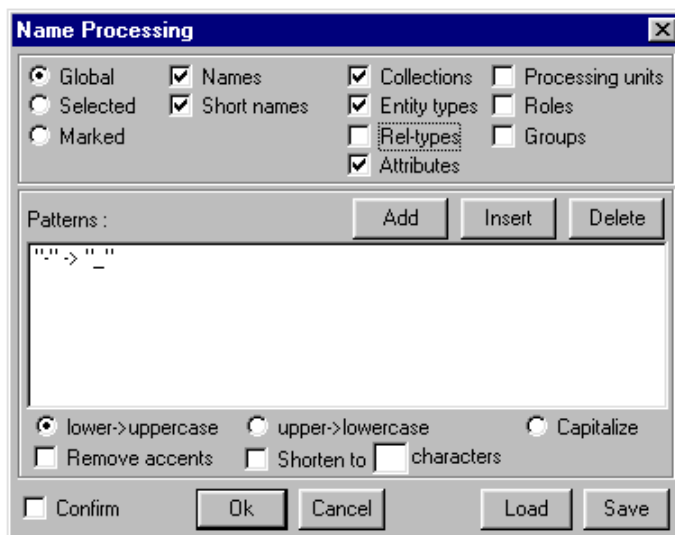


Figure 6.22 - Processing the names of the schema.

We proceed as follows:

- we set the scope to Global (i.e., processing the whole schema);
- we want to process both the Names and the Short names ...
- ... of the Entity types, Attributes and Collections;
- first, we tell the processor that we want all the names to be converted into uppercase characters (button lower -> uppercase)
- then we define the translation pattern:
 - we click on button Add, therefore opening the New pattern box (Figure 6.23):
 - the character - is typed in the Search for field,
 - the character _ is typed in the Replace by field,
 - and we confirm by clicking on the button OK;

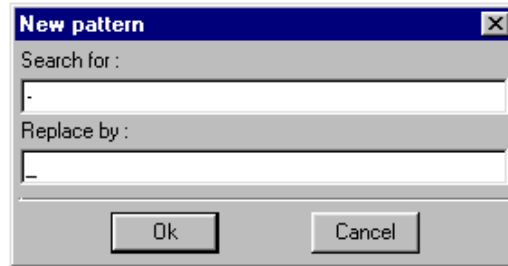


Figure 6.23 - Defining a substitution pattern.

- the translation pattern "-" -> "_" now appears in the Patterns field (Figure 6.22);
- we leave the button Confirm unchecked to avoid being asked for confirmation before each substitution;
- we validate by clicking on the button OK.

All the "-" characters have been replaced with the character "_", just as we wanted them to be and all the names are now in uppercase (Figure 6.24).

The same procedure will also be used to remove space characters or to replace the reserved words it may comprise: no user name can belong to a list that includes such words as *create*, *table*, *integer*, *char*, *date*, *index*, *references*, *unique*, *check*, etc.

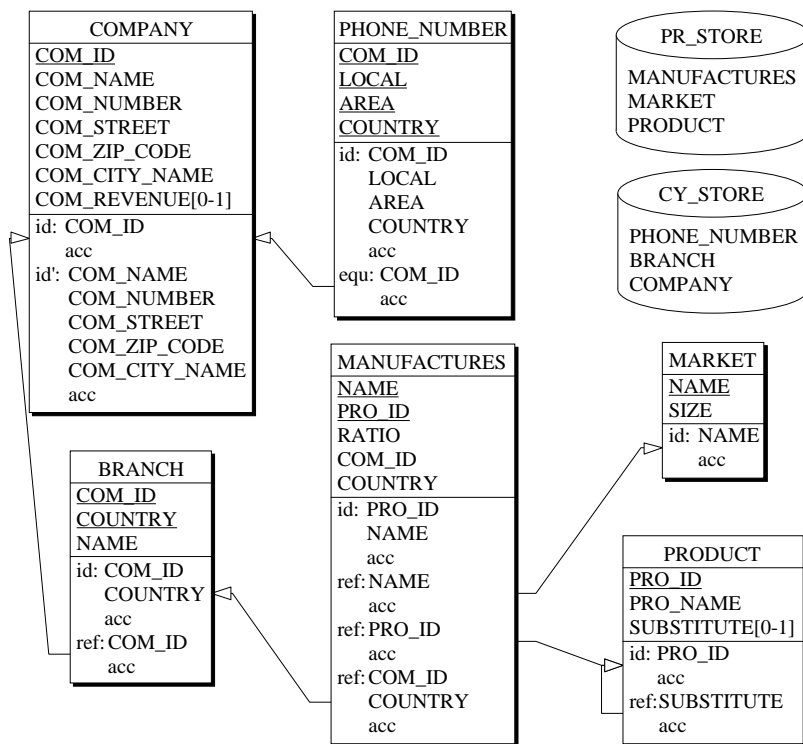


Figure 6.24 - The final schema.

6.9 SQL code generation

Now we can ask for the SQL translation function as proposed in lesson 4 through the command **File / Generate / Standard SQL(check)**.

```
-- *****
-- * Standard SQL generation *
-- *-----*
-- * Generator date: Dec 8 1998 *
-- * Generation date: Mon Jan 04 21:50:31 1999 *
-- *****
```

```
-- Database Section --

create database Manufacturing;

-- DBSpace Section --

create dbspace CY_STORE;
create dbspace PR_STORE;

-- Table Section --

create table BRANCH (
    COM_ID char(15) not null,
    NAME char(20) not null,
    COUNTRY numeric(3) not null,
    primary key (COM_ID,COUNTRY))
in CY_STORE;

create table COMPANY (
    COM_ID char(15) not null,
    COM_NAME char(25) not null,
    COM_NUMBER numeric(5) not null,
    COM_STREET char(20) not null,
    COM_ZIP_CODE numeric(7) not null,
    COM_CITY_NAME char(18) not null,
    COM_REVENUE numeric(12),
    primary key (COM_ID),
    unique (COM_NAME,COM_NUMBER,COM_STREET,COM_ZIP_CODE,
           COM_CITY_NAME))
in CY_STORE;

create table MANUFACTURES (
    PRO_ID char(8) not null,
    NAME char(24) not null,
    RATIO numeric(4,4) not null,
    COM_ID char(15) not null,
    COUNTRY numeric(3) not null,
    primary key (PRO_ID, NAME))
in PR_STORE;

create table MARKET (
    NAME char(24) not null,
    SIZE numeric(6) not null,
    primary key (NAME))
in PR_STORE;
```

```
create table PHONE_NUMBER (  
    COM_ID char(15) not null,  
    LOCAL numeric(8) not null,  
    AREA numeric(3) not null,  
    COUNTRY numeric(3) not null,  
    primary key (COM_ID, LOCAL, AREA, COUNTRY))  
in CY_STORE;  
  
create table PRODUCT (  
    PRO_ID char(8) not null,  
    PRO_NAME char(25) not null,  
    SUBSTITUTE char(8),  
    primary key (PRO_ID))  
in PR_STORE;  
  
-- Constraints Section  
-- _____  
  
alter table BRANCH add constraint FKBELONGS  
    foreign key (COM_ID) references COMPANY;  
  
alter table COMPANY add constraint  
    check(exists(select * from PHONE_NUMBER  
        where PHONE_NUMBER.COM_ID = COM_ID));  
  
alter table MANUFACTURES add constraint FKMAN_MAR  
    foreign key (NAME) references MARKET;  
  
alter table MANUFACTURES add constraint FKMAN_PRO  
    foreign key (PRO_ID) references PRODUCT;  
  
alter table MANUFACTURES add constraint FKMAN_BRA  
    foreign key (COM_ID,COUNTRY) references BRANCH;  
  
alter table PHONE_NUMBER add constraint FKCOM_PHO  
    foreign key (COM_ID) references COMPANY;  
  
alter table PRODUCT add constraint FKREPLACES  
    foreign key (SUBSTITUTE) references PRODUCT;  
  
-- Index Section --  
  
create unique index IDBRANCH on BRANCH (COM_ID,COUNTRY);  
create index FKBELONGS on BRANCH (COM_ID);
```

```
create unique index IDCOMPANY on COMPANY (COM_ID);
create unique index IDCOMPANY on COMPANY (COM_NAME,COM_NUMBER,
COM_STREET,COM_ZIP_CODE,COM_CITY_NAME);
create unique index MANUFACTURES on MANUFACTURES (PRO_ID,NAME);
create index FKMAN_MAR on MANUFACTURES (NAME);
create index FKMAN_PRO on MANUFACTURES (PRO_ID);
create index FKMAN_BRA on MANUFACTURES (COM_ID,COUNTRY);
create unique index IDMARKET on MARKET (NAME);
create unique index IDPHONE_NUMBER on PHONE_NUMBER (COM_ID,LOCAL,
AREA,COUNTRY);
create index FKCOM_PHO on PHONE_NUMBER (COM_ID);
create unique index IDPRODUCT on PRODUCT (PRO_ID);
create index FKREPLACES on PRODUCT (SUBSTITUTE);
```

Figure 6.25 - The SQL program creating the database.

About the coding rules


The rules used to transcript relational structures into SQL statements are rather straightforward. Two comments are worth making:

- Foreign keys have been created in a specific section that follows the table creation statements. The reason is that most SQL compilers do not accept forward references, i.e., declaring foreign keys whose target table has not been declared yet.
- The second constraint declaration certainly will have drawn your attention. Remember that the `equ` constraint between `PHONE_NUMBER.COM_ID` and `COMPANY.COM_ID` is made of two independent constraints, namely a foreign key from `PHONE_NUMBER` and an inclusion constraint from `COMPANY.COM_ID` and `PHONE_NUMBER`. This declaration defines the predicate that expresses the second part of the `equ` constraint.

In fact, **this translation may not work** for two reasons. First, many DBMS do not accept `check` predicates referencing more than one row. Secondly, the translation of the two parts of the `equ` constraint form a kind of **deadlock**. Indeed, they imply that (1) a `PHONE_NUMBER` row cannot be inserted before its parent `COMPANY` row has been inserted, and (2) a `COMPANY` row cannot be inserted before the first of its children `PHONE_NUMBER` rows has been inserted! The solution lies in specific transaction structures based on deferred constraints. These techniques go well beyond the scope of this tutorial.

On SQL generation styles

Despite what this section may suggest, there are many ways to code a logical/physical schema into a SQL program. Actually, coding all aspects of a logical schema can prove a complex task⁷. DB-MAIN proposes several styles from which you can choose.

- *Built-in generators.* The **File / Generate command** proposes 6 simple SQL generators. Explore them to evaluate their characteristics. In particular, what do you think of the **Academic** style?
- *External generators.* They are available as *.OXO programs through the **File / Execute Voyager** command or the button . Program SQL.OXO is similar to built-in generator **Standard SQL (check)**. The source code of this generator is available as file SQL.V2, and can be modified.
- *Component SQL-GEN of Application Library #1.* This collection of 9 generators proposes different ways to code basic constructs such as identifiers and foreign keys.
- A more comprehensive and parametrized generator is available, but its use requires concepts that go beyond the scope of this tutorial.

In this section, we have mentioned a key feature of DB-MAIN, namely the Voyager extensions. Though this part of the tool is too complex to be discussed in this volume, we will say some words on its characteristics.

The Voyager 2 meta-development environment

DB-MAIN offers a complete programming language, *Voyager 2*, that allows analysts to develop their own components to include in the CASE tool. Sophisticated code and report generators, but also analyzers, transformers, evaluators, etc., can be developed without resorting to the C++ native language of DB-MAIN. Programs written in *Voyager 2* (*.V2) are compiled into *.OXO binary code, that is executed by the Voyager virtual machine of DB-MAIN.

SQL.OXO and the components of the *Application Libraries*⁸ are some examples of public-domain extensions.

The Voyager development environment is described in specific manuals.

7. In fact, there are currently *no good commercial SQL generators*, i.e., generators that produce a correct and efficient code for all the constructs of logical schemas. In addition, the current generators are rigid code builders that allow for practically no customization.

6.10 About the DB-MAIN graphical representation

One could be surprised by the graphical conventions used to represent identifiers and reference groups. Indeed, such software packages as Microsoft Access also represent relational schemas in a graphical way through simpler techniques (Figure 6.26). Primary keys are highlighted and references are shown through directed field-to-field connections.

The reason lies in the greater generality of CASE tools, and particularly of DB-MAIN. Such tools must cope with much more complex situations, not only in developing new databases according to relational and non-relational technologies, but also in describing the recovery of ancient databases that exhibit non-standard structures. For instance, consider how simple graphical drawing conventions as those used in Figure 6.26 could cope with the following situations:

- a table can have an arbitrary number of identifiers;
- two identifiers can share common columns (without one of them being non-minimal);
- a foreign key can reference secondary identifiers;
- an identifier can be a foreign key as well;
- two foreign keys can share common columns;
- a foreign key can reference more than one table;
- self-referencing tables (a foreign key of table T references T);
- the components of an identifier or of a foreign key can be submitted to additional constraints, such as *coexistence* (will be seen later);
- two foreign keys submitted to constraints such as *coexistence* or *exclusivity* (will be seen later);
- an identifier (or a foreign key) can have specific properties that make it an object on its own: semantic description, usage statistics, physical parameters, implementation technology, etc.

These constructs, and others, can only be described by making identifiers, foreign keys, and in general any group, specific objects, with their own graphical representation.

-
8. An *Application Library* is a collection of useful Voyager/Delphi applications that can be used by analysts. The first volume includes executable and source versions of two RTF report generators, a natural language paraphraser, a schema metrics evaluator, a performance evaluator for relational and COBOL databases, and an organizational units management system for data administration.

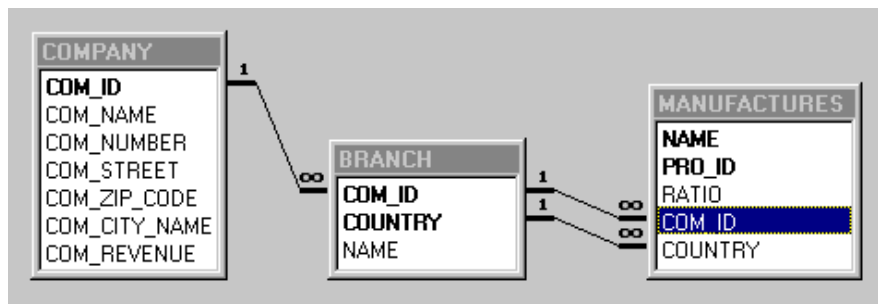


Figure 6.26 - A (seemingly) more intuitive way to represent primary and foreign keys (Relation window of Microsoft Access).

6.11 Logical vs physical schemas

Now it is time to give some explanation of the title of this lesson. Actually, what is exactly a *logical schema* and what is a *physical schema*?

The answer is not simple since it depends on the nature of the DBMS.

- For most authors, a **logical schema** is a non-technical description of the database according to the model of a family of DBMS. A schema that includes the description of tables, columns, primary keys and foreign keys is a *relational logical schema*. In the same way, a schema describing files, record types, fields and record keys can be called a *COBOL logical schema*.

We will add another property: a logical schema is the necessary and sufficient information a programmer or a user must be supplied with in order to write queries and programs that use the database. This means that any construct whose knowledge is not required to write programs does not belong to the logical schema of the database. For instance, indexes (access keys) and storage spaces (collections) are not part of relational logical schemas. On the contrary, these constructs are integral parts of COBOL logical schemas, since the programmer must explicitly mention them in his/her programs.

- The **physical schema** of a database includes constructs and properties that govern the placement of data in the secondary memory, as well as the way they are accessed and updated. These specifications are strongly performance-oriented and require from the database engineer a deep knowledge of the technical aspects of the H/S environment and of the DBMS. They can

be ignored by the programmers. It is clear that indexes and storage spaces, as well as page size and buffer management are components of the *relational physical schema*.

Now let us return to the lesson title. To simplify the discussion, we have not distinguished between logical and physical components of relational schemas. Therefore, all the schemas of this lesson are mixed logical/physical, since they include both logical and physical constructs. In Volume 2, devoted to *Information analysis and database design*, we will carefully separate these schemas as well as the reasonings and techniques that relate to them.

6.12 Closing the lesson

We can now quit DB-MAIN. The modified project can be saved as suggested by DB-MAIN.

Key ideas of Lesson 6




1. **Reminder.** A **conceptual schema** describes the abstract concepts that are (or will be) represented in a database. It is made of entity types, attributes, rel-types, identifiers and various other constructs.
2. A **logical schema** is the description of the database according to the model of a family of DBMS. A *relational logical schema* mainly describes tables, columns, primary keys, candidate keys and foreign keys. Users and programmers need to consult the logical schema to write queries and programs running on the database. A logical schema is obtained by applying representation rules to the conceptual constructs. For instance, an entity type can be represented by a table, an attribute by a column, a functional rel-type by a foreign key and a complex rel-type by a relationship table. Names must often be translated as well. More complex rules can be designed to meet advanced criteria.
3. A **physical schema** includes the data structures of the logical schema, enriched with technical constructs defining the implementation and exploitation modes of the physical database. Physical schemas are strongly dependent on the specific DBMS with which the database is implemented. Its design is mainly performance-oriented. A typical relational physical schema includes, among others, the specification of storage spaces and indexes.
4. A **coded schema** is a program expressed into the Data Description Language of a DBMS (such as SQL-DDL). Coding rules are straightforward for the main constructs of the physical schema. However, some constraints can lead to complex and tricky code that make the coding process a far from trivial task.
5. A **reference group** (generally called *foreign key* in relational DBMS) is a group of attributes (columns) whose values are used to designate rows in another (or in the same) table.
6. An **access key** (generally called *index* in relational DBMS) is a group of attributes (columns) which an access mechanism is associated with, that provides fast access to records (rows) matching the values of this group.
7. An **entity collection** is a named storage area in which records or table rows can be stored. It corresponds to files or table spaces in most DBMS.

Summary of Lesson 6

- In this lesson, we have studied new notions:
 - ref reference group (or foreign key)
 - equ reference group
 - access key (e.g., index)
 - entity collections

- We have also discussed in further detail the concepts of *logical schema*, *physical schema* and *coded schema*, as opposed to conceptual schemas.

- We have learned about *Voyager* external programs.

- We have learned,
 - to define a group **New / Group** 
 - to define a reference group
 - the **Constraint** button in the Group Property box
 - to define an access key
 - the **Access key** button in the Group property box
 - to define a collection
 - New / Collection** 
 - to replace substrings in names
 - Transform / Name processing**
 - to execute an external *Voyager* program
 - File / Execute Voyager** 

Exercises for Lesson 6

- 6.1 Enter manually⁹ a relational logical schema describing the database that was built by the following SQL program:

```
create database RESULTS;

create table STD (
    STD_ID char(15) not null,
    STD_NM char(25) not null,
    STD_PHONE char(10),
    primary key (STD-ID) );

create table LCT (
    LCT_CD char(5) not null,
    LCT_NM char(25) not null,
    primary key (LCT_CD) );

create table CRS (
    CRS_NM char(25) not null,
    LCT_CD char(5) not null,
    HOURS decimal(3) not null,
    primary key (CRS_NM,LCT_CD),
    foreign key (LCT_CD) references LCT) );

create table RES (
    STUD_ID char(15) not null,
    CRS_NM char(25) not null,
    LCT_CD char(5) not null,
    GRADE decimal(5,1),
    primary key (STUD_ID,CRS_NM,LCT_CD),
    foreign key (STUD_ID) reference STUD,
    foreign key (CRS_NM,LCT_CD) references CRS )
);
```

9. Frustratingly (for you!), DB-MAIN includes a powerful tool that can build logical schemas from SQL code. However, using it would make you miss the objective of the exercise.

- 6.2 This schema is particularly obscure, due to the choice of (too) short names. In fact, the names can be changed to make them more informative. Applying the following substitution leads to a much more readable schema:

```
STD → STUDENT
LCT → LECTURE
CRS → COURSE
RES → RESULT
NM  → NAME
CD  → CODE
```

Use the Name processing function to carry out these replacements. Note that you can add several patterns in the `PATTERNS` field, so that all the transformations can be executed in one operation.

- 6.3 Define the access keys (applying **Transform / Relational model** will do the job), then generate a new SQL creation program. Though structurally equivalent to the first one, it enjoys a highly desirable quality: readability.
- 6.4 Try to guess which conceptual schema this logical schema could have come from¹⁰.
- 6.5 Consider Project MANU-6 again. Rework the schema hierarchy and some schema constructs in order to propose a neater organization:
- the hierarchy shows the conceptual, logical, physical and coded schemas;
 - the physical schema does not include *prefix access keys*.

10. Note that this kind of problem resorts to the Database Reverse Engineering domain, which will be addressed later on..

Lesson 7

Names

Objective

This lesson stresses the importance of names. It discusses constraints on names in a schema and compares different approaches to assign names to objects. It also describes three name processors: one that can translate, transform and convert names (*Name processing*), a second one that changes the prefix of a series of names (*Change prefix*) and a final one that manages consistent sets of synonyms (*Lexicons*).

7.1 Introduction

Names are the main links between the abstract and formal objects of a schema and the application domain objects. So, the quality of a schema strongly depends on the way names have been assigned to objects. In addition, object names can change from a schema to another one. For instance, conceptual names are not convenient to generate SQL programs. Hence the need for changing names. All this is the topic of this lesson.

7.2 Uniqueness rules

When entering and modifying schemas in former lessons, you certainly have observed that you cannot give objects arbitrary names. Obviously, some rules must be followed, such as *the entity types of a schema have distinct names*. Indeed, the DB-MAIN model includes naming constraints that make it possible to denote objects through their name. Here are the main rules:

- Two names composed of the same characters, be they in uppercase or in lowercase, in the same order, are considered identical; so, Customer and CUSTOMER are the same names; the accents are taken into account, so that Elève and Elevé are distinct names;
- all the printable characters, including spaces, /, [, {, (, punctuation symbols and diacritical characters, can be used to form names; however symbols " and | are prohibited;
- the *schemas* of a project are identified by the combination <name>/<version>, or merely <name> if <version> is empty;
- each *entity type* of a schema is identified by its name;
- each *rel-type* of a schema is identified by its name;
- a *collection* of a schema is identified by its name;
- direct *attributes* of a definite parent (an entity type, a rel-type or a compound attribute) have distinct names;
- a *group* of a definite parent (idem) is identified by its name.
- each *processing unit* of a definite parent (an entity type, a rel-type or a schema) is identified by its name.

We can enforce stricter rules through the *Schema analysis assistant* that will be discussed later on. For example, we can stipulates that attribute names are distinct throughout the schema.

7.3 Ambiguous names (the | symbol)

The standard uniqueness rules described above may appear too strong in some situations, particularly for rel-types. For instance, the analyst who builds a tree-like schema (Figure 7.1) may find it useless to name rel-types.

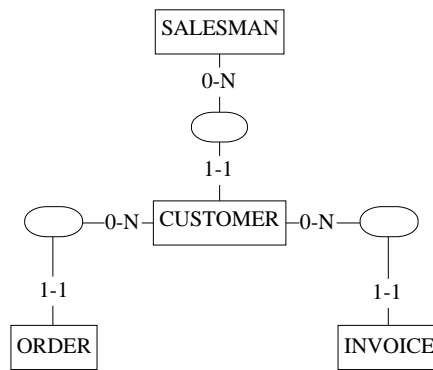


Figure 7.1 - A hierarchical schema (IMS-like¹) in which rel-types are left unnamed, without loss of readability.

There exist other variants of Entity-relationship conceptual models, such as NIAM or ORM (Object-Role)², that insist on role names but ignore rel-type names. Moreover, many schemas include a large number of rel-types defining generic relations such as "part of", "in", "of", "cross", "overlap", etc. In these situations the analyst would want to give these rel-types either the same name (Figure 7.3), or no name at all (Figure 7.6). The syntax of DB-MAIN names includes the special symbol "|", which is a valid character, but which has a special effect when displayed in a schema view: *this character as well as all the characters that follow it are not displayed*.

Figure 7.2 shows a schema in which three entity types are being given the same name. The full name of the current entity type includes a visible part

-
1. In *hierarchical* databases records are organized into tree structures: a record either is a root record (no parent) or is a child record that has one parent record. IMS from IBM is the main DBMS of this category. The term *hierarchical* is standard, but a bit misleading, since hierarchically organized records can have an arbitrary number of parents.
 2. These models are at the core of specific approaches of information system design based on a linguistic analysis. They will be discussed later on.

"PERSON" and a hidden part "| version 3". No other entity types can bear the name "PERSON| version 3". However, several entity types can be assigned the same visible part, as shown in the right list box.

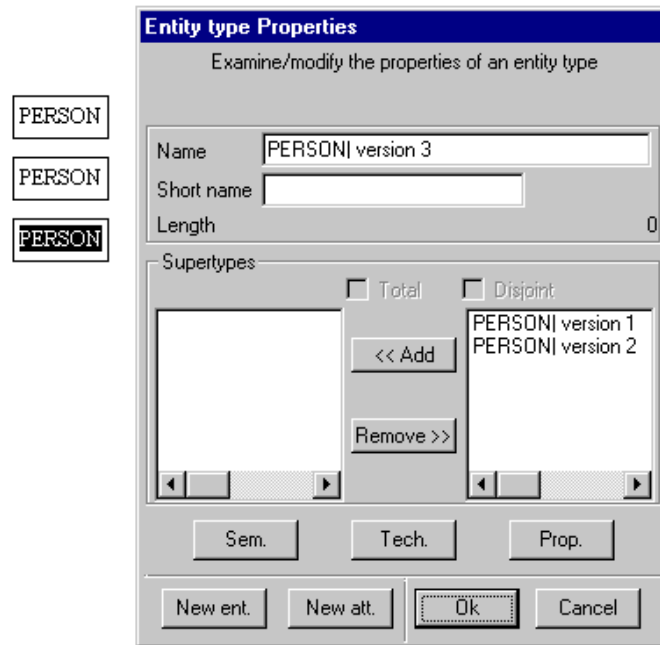


Figure 7.2 - The full name of the current entity type includes a visible part "PERSON" and a hidden part "| version 3".

7.4 How to choose names

Models, methodologies and CASE tools generally offer the analysts (almost) full freedom to give names to objects in a schema. However, giving objects quite arbitrary names would lead to a poor schema, which will be difficult to read, to interpret and to use.

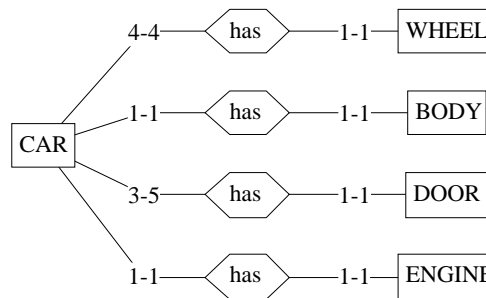


Figure 7.3 - It would be worthless to give these rel-types distinct names such as "has wheels", "has body", etc. In fact, they have been given *distinct* names "has | wheels", "has | body", etc.

So, the question of how to assign names to objects in a consistent way is quite relevant.

In most cases, **entity types** represent *major concepts* of the application domain, so that it is natural to name each of them with the **noun** of the concept: CUSTOMER, ACCIDENT or BRANCH. Whether it is better to use singular or plural forms is a matter of taste. In this book, we use the singular form to denote the concept more than its population: the *archetypal customer* instead of the *set of customers*.

Attributes denote *local properties*, and are given names that suggest these properties. Generally, attribute names are **nouns**, such as Date, Amount, Address. In some cases, the name can take a more complex form, such as an assertion: HasChildren, IsValid, IsComingFrom. Many of these attributes have a boolean domain.

Naming **rel-types** and **roles** can be more complex. Let us first observe that they do not always require names, as illustrated in Figure 7.1. In many cases, a rel-type represents an *action* between two concrete or abstract concepts: a customer *signs* a contract, a company *rents* cars, an order *has* details, an accident *involves* vehicles, etc. Therefore, many rel-types are given names which derive from **verbs**, such as the verb itself (Figure 7.4) or an abbreviated form of it (*from* instead of *coming from* in Figure 7.5). According to this approach, roles are given names that denote the subset of the entity type that plays this role.

Rel-type *rents* involves two participants, namely the *rented car*, which is a car, and the *renter*, which is a company. Hence role names *rented car* and *renter* (Figure 7.4). **Roles** generally are given no names (in which case the name of the entity type is used instead), or names which are **nouns**. Though it is quite neutral as far as naming conventions are concerned, the DB-MAIN model slightly favors these rules as can be observed in the composition of groups: a group comprises components that are best denoted by nouns (sender and SerNumber in Figure 7.5).

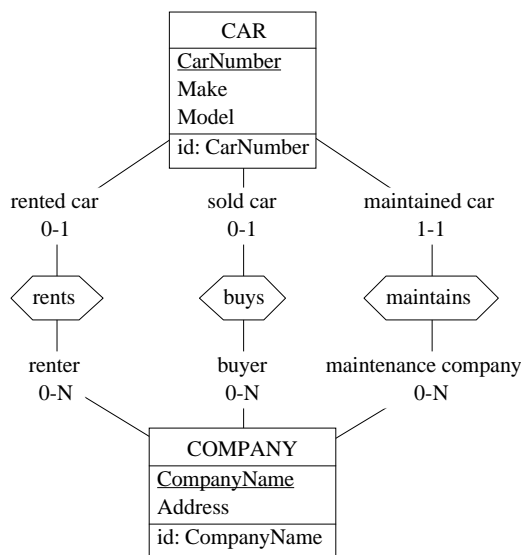


Figure 7.4 - Conventional naming conventions. Generally, roles are given no names, except to solve ambiguities in cyclic rel-types.

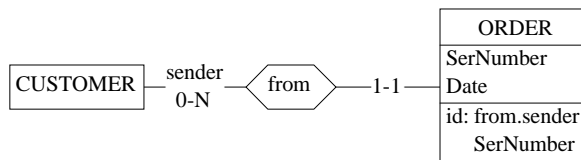


Figure 7.5 - Use of role names in groups.

This convention for naming rel-types has a drawback though. Since the name of the rel-type is the verb extracted from the assertion that defines its semantics, the verb can have two forms, namely active or passive. For instance, the semantics of rel-type `rents` of Figure 7.4 can be expressed as *a company rents cars*, or, equivalently, as *cars are rented by companies*, hence two possible names for this rel-type: `rents` and `rented by`. To avoid this problem, some authors use the infinitive form of the verb, i.e., *to rent* in our case. By choosing the correct form of the verb, the rel-type can then be read either way.

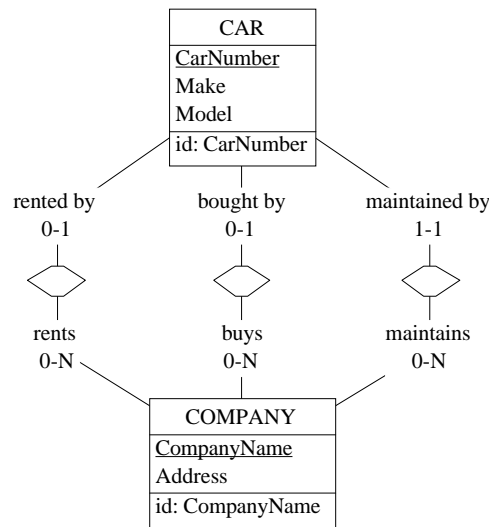


Figure 7.6 - Rel-type and role naming conventions that use roles to name the relationships between entity types. Such conventions will be found in ORM models, but can be used in Entity-relationship schemas as well.

As mentioned above, some approaches are based on roles more than on rel-types. Such is the case for ORM³, in which each role is given a name that allows users who read a schema to form natural sentences.

For instance, the schema of Figure 7.6 can be read as follows:

- *a car is rented by a company*

3. The Object Role Model, first developed by G. Njissen in the eighties, has since been extended by several authors. See [Halpin,1996] for instance.

- a company rents cars
- a car is maintained by a company
- a company maintains cars
- etc.

Tentative conclusion

There is no unique naming approach that brings all benefits but no problems in all situations. The important point is that the naming conventions you adopt must be **consistent** throughout your schemas.

7.5 Name processing

For different reasons, it can be useful or even necessary to modify the names in a schema. For example, names must be in uppercase, or must not be more than N character long, or cannot include some substrings. Processing each name individually can be realistic for small schemas, but coping with thousands of names cannot be performed without tools. The *Name processing* tool of DB-MAIN has been developed with this objective. This tool is not quite new for us. Indeed, we already used it in Lesson 6 to replace characters "-" with "_" in physical schemas.

The main control panel is shown in Figure 7.7. It includes the following sections:

1. **Scope.** Defines the objects to which the transformations will be applied.
 - *Where:* Global: in the whole schema; Selected: among the selected objects; Marked: among the marked objects.
 - *Which names:* Names and/or Short names.
 - *For which object types:* Collections, Entity types, Rel-types, etc.
2. **Substitution patterns.** List of substitution patterns in the following format:
 - "search string" → "new string"
3. **Character transformations.**
 - change case, capitalize, remove accents and shorten names

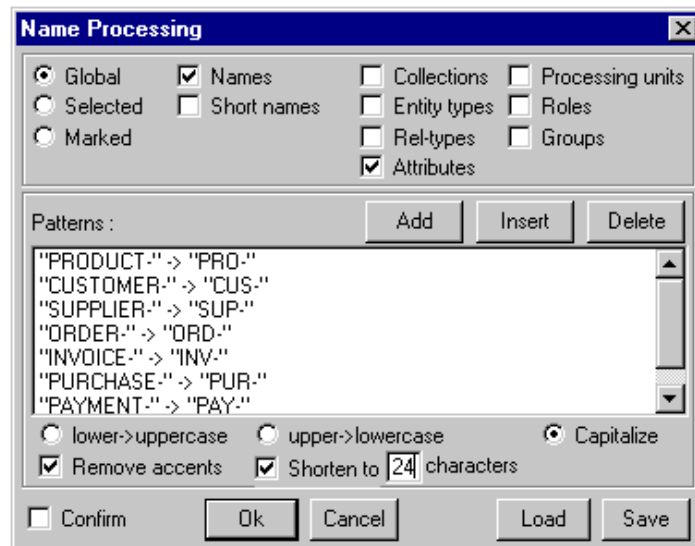


Figure 7.7 - This panel has been programmed to normalize the names of the attributes of the current schema. The accents are removed, names are capitalized (first character in uppercase, the others in lowercase), and long object names are replaced with short names. In addition, the resulting names are trimmed to 24 characters.

4. Load/Save control parameters.

- loads/saves the contents of the panel, including the substitution patterns. Saved in a *.pat file.

The principles of pattern substitution are simple. Consider the form as it appears in the pattern field of the panel (note that quotation marks are not part of the strings and must not be typed):

"search string" → "new string"

1. Each instance of *search string* found in the defined scope is replaced with the string *new string*.
2. It is possible to tell that *search string* are the *first characters* (with symbol ^) or the *last characters* (with symbol \$) of the searched names.
3. "old string" can include wildcard characters * and ?. ? matches any single character while * matches any empty or non-empty string.

4. If *new string* is empty, then the instances of *search string* are deleted.

To illustrate the use of substitution patterns, let us consider the pattern

$$\text{search} \rightarrow \text{"XXX"}$$

where *search* is the search argument that will be discussed. Let "elephant" be the name to process. Considering various substitution patterns, the results will be as shown in Figure 7.8.

substitution patterns	resulting name
"leph" → "XXX"	eXXXant
"e" → "XXX"	XXXlXXXphant
"e?" → "XXX"	XXXXXXhant
"*p" → "XXX"	XXXhant
"l*h" → "XXX"	eXXXant
"p*" → "XXX"	eleXXX
"e*" → "XXX"	XXX
"*" → "XXX"	XXX
"^e" → "XXX"	XXXlephant
"t\$" → "XXX"	elephanXXX
"e\$" → "XXX"	elephant
"^e*p?a*t\$" → "XXX"	XXX

Figure 7.8 - Applying substitution patterns to the name elephant.

The list of substitution patterns can be as long as you want. For instance, you can define a translation dictionary to convert the names of a schema into another language. However, do not expect a high quality translation when composed names are frequent!

When more than one transformation is asked for, they are performed in the following order:

1. pattern matching
2. removal of accents
3. case conversion
4. shortening

7.6 Changing the prefix of names

A nice little tool allows you to change the prefix of a series of attribute names. These attributes are all the direct attributes of a parent object which can be an entity type, a compound attribute or a rel-type. Proceed as follows (Figure 7.9).

1. Select the parent object (here PRODUCT).
2. Execute function **Transform / Change prefix**.
3. The prefix processor computes the largest prefix of the attributes of the parent objects and displays it in the Prefix field (PRO-). This prefix can be empty.
4. Change this value in the Prefix field (type P_). If you want to suppress this prefix, empty the Prefix field.
5. Click on OK.

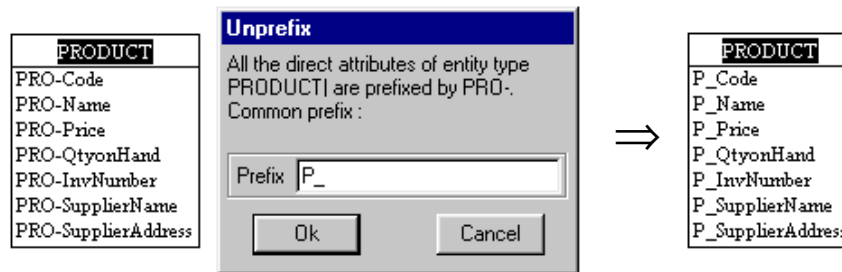


Figure 7.9 - The largest prefix ("PRO-") of the names of direct attributes of entity type PRODUCT have been replaced with the new prefix "P_".

7.7 Lexicons

Each object in a schema, including the schema itself, receives a name. To be more precise, it received **only one name**⁴. Very often, one wants to give an object several names, generally called *synonyms*. However, giving some objects two names, while giving another one three or four names quickly leads

-
4. The *short name* that appears in the property box of several objects and in the *Text extended view* cannot be considered as a name in its own right. It is mainly used as an aid for building names automatically in some processes, such as transformations and generators.

to an unmanageable situation. The DB-MAIN tool proposes a disciplined way to assign synonyms to objects through the concept of **Lexicon**.

A Lexicon is a consistent set of names, each one being assigned to one object of the schema. Figure 7.10 shows the same schema to which two lexicons have been applied. The first one gives objects English names (left), while the second one gives them French names (right).

Creating the *English Lexicon* is straightforward:

1. We assign each object an English name;
2. We create a new Lexicon with the name "English".

To create another Lexicon, we proceed in the same way:

1. We update each object to give it a French name;
2. We create a new Lexicon with the name "French".

Similarly, we can define *Dutch, German, Italian or Arabic* Lexicons. Lexicons can be used to give objects *natural names, normalized names, technical names*. They can be used to assign *COBOL, Java or RPG* names to objects.

To give objects the names from a definite Lexicon, *display the Lexicon*. An existing Lexicon can be *updated* and *deleted*.

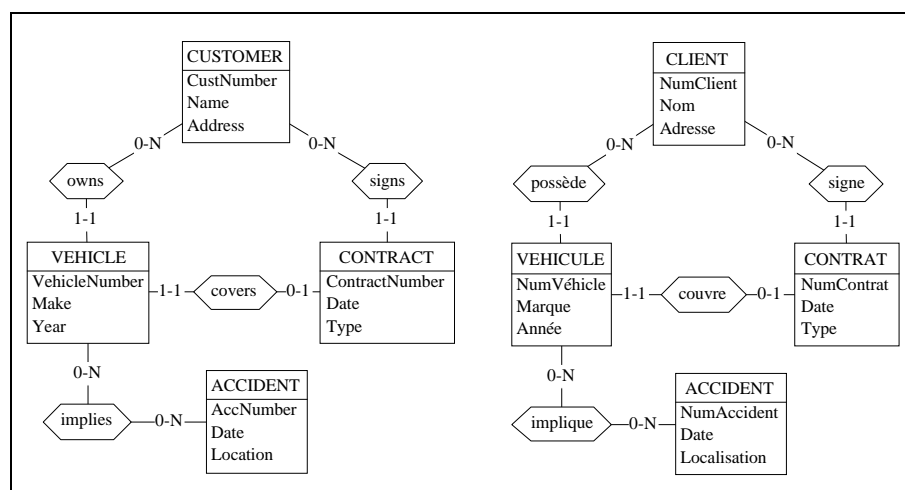



Figure 7.10 - The same schema displayed according to two distinct Lexicons, respectively called English and French.

The Lexicons are managed by a small *Voyager 2* application⁵ called `Lexicon.oxo`. It can be run as follows:


1. click on the red button  in the Standard tools bar (or use command **File / Execute Voyager**)
2. double-click program file `Lexicon.oxo`.

The Lexicon processor proposes four functions:

1. *Create a Lexicon*: asks you a Lexicon name and stores all the current names in it;
2. *Update a lexicon*: shows all the known Lexicons, lets you choose one and stores all the current names in the selected Lexicon;
3. *Display a Lexicon*: shows all the known Lexicons, lets you choose one and replaces the current names with the contents of the selected Lexicon;
4. *Delete a Lexicon*: shows all the known Lexicons, lets you choose one and deletes it.

Type the number of the selected function into the bottom field, click on OK and follow the instructions.

Note

If you want to run the `Lexicon` program again, you do not need to proceed according the above procedure. Indeed, this program still is in the memory of the *Voyager* abstract machine. All you need to do to run it is to click on the button  (or to execute command **File / Rerun Voyager**).


5. We already met *Voyager* applications in Lesson 6, where we mentioned that `DB-MAIN` can be enriched by additional components developed in the meta-language *Voyager 2*.

Key ideas of Lesson 7

1. Names are essential components of schemas. Carefully choosing them is an important task that will make schemas easy to use by both users and developers, therefore maximizing the reliability of the specifications.
2. A name is used to designate an object among all the objects of the same kind, in a definite scope, such as *all the entity types of the schema* or *all the direct attributes of an entity type*.
3. Generally, a name is made of a non-empty character string. However, giving empty names to rel-types and roles can be useful.
4. In many cases, the following naming conventions yield schemas that are easy to read: *name an entity type* with the noun with which the domain concept is referred to; *name an attribute* with the noun with which the domain property is referred to; *name a rel-type* with the verb with which the domain relationship is referred to. Give *roles* the name of the subset of the entity type that play this role. Other consistent rules can be chosen to name objects in a schema.
5. In several engineering processes, it can be useful to modify the names of the objects of a schema to improve their consistency or their readability, or to make them compliant with the syntax of a DBMS. The *Name processing* tool and the *Change prefix* function of DB-MAIN can help changing names in a large and complex schema.
6. Assigning several names to an object is an important requirement in some environments. The DB-MAIN Lexicon manager can be used to create sets of synonyms for the objects of a schema.

Summary of Lesson 7

- In this lesson, we have studied new concepts:
 - ambiguous names; visible and hidden parts of names
 - *Lexicons* as sets of consistent synonyms for the objects of a schema

- We have also learned:
 - to choose names for objects in a schema
 - to define ambiguous names: symbol |
 - to use rel-type and role names to describe relationships between application concepts
 - to change the names in a schema:
 - Name processing tool
 - Change prefix tool
 - to define and use Lexicons:
 - Lexicon.oxo Voyager processor
 - to rerun loaded *Voyager* processors:
 - File / Rerun Voyager** 

- We have produced a new type of file:
 - Name processing parameters (* .pat).

Exercises for Lesson 7

- 7.1 Open the schema of Figure 5.15 you built in Lesson 5. Define a set of substitution patterns that give other names to all the objects of this schema. For instance, change *City* into *Town*, *Size* into *Volume* and *manufactures* into *produces*. Check it on a copy of the schema.
- 7.2 Define *Name processing* parameters that translate the names in the schema of Figure 5.15 in such a way that the derived relational schema will meet the following requirements:
 - names are SQL-compliant (no "-", no SQL reserved words);
 - names are in uppercase;
 - names have at most 10 characters.
- 7.3 Considering the origin names in Figure 5.15 and the sets of names defined above, build three Lexicons.
- 7.4 In the play *The Bald Soprano* (*La cantatrice chauve*) playwright Eugene Ionesco imagines a scene in which two characters talk about their relatives (parents, neighbors, doctors and even dogs), who all happen to be named *Bobby Watson*!
Build a schema in which all the objects (entity types, rel-types, attributes) are named Bobby Watson.

Lesson 8

More about entity types

Objective

This lesson discusses the concepts of supertype/subtype relation (also called *IS-A* relation), of total/partial and exclusive/overlapping subtypes, and of inheritance. These constructs define a classification scheme according to which an entity type can belong to more than one type by declaring it a subtype of another one, thus inheriting some of its properties from the latter. Procedures can be associated with entity types, rel-types and schemas. Such procedures (or methods) can be used to define object classes in object-oriented schemas.

8.1 Starting Lesson 8

We start DB-MAIN and we create a new project called, say, Lesson08.

8.2 Classification hierarchies (IS-A relations)

We create a new schema with name ISA and version 1.

Let us suppose that we are describing the activities of *factories* which are in relation to their *suppliers* and their *customers*, which all are *companies*.

In other words, factories, suppliers and customers are companies. In addition, each factory can have customers and can have suppliers. From now on however, we will ignore the latter facts.

If we represent factories, suppliers, customers and companies by entity types FACTORY, SUPPLIER, CUSTOMER and COMPANY respectively, we get the schema of Figure 8.1.

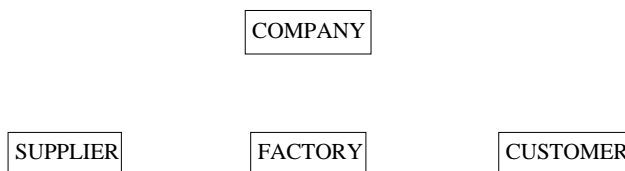


Figure 8.1 - Four unrelated entity types (so far!).

We then have to express some additional facts:

- a factory is a company as well;
- similarly, each supplier is a company;
- and each customer is a company.

Another way to describe these facts is to say that a factory (as well as a supplier and a customer) is a *special kind of* company. This translates in the Entity-relationship model as follows:

- FACTORY is declared a **subtype** of COMPANY;
- SUPPLIER is a **subtype** of COMPANY;
- CUSTOMER is a **subtype** of COMPANY.

Conversely, we can say that COMPANY is a **supertype** of FACTORY, SUPPLIER and CUSTOMER.

To define this *subtype/supertype* relation, we open the Entity box of FACTORY (double-click as usual), and we move the name COMPANY from the right list to the Supertypes list on the left (Figure 8.2).

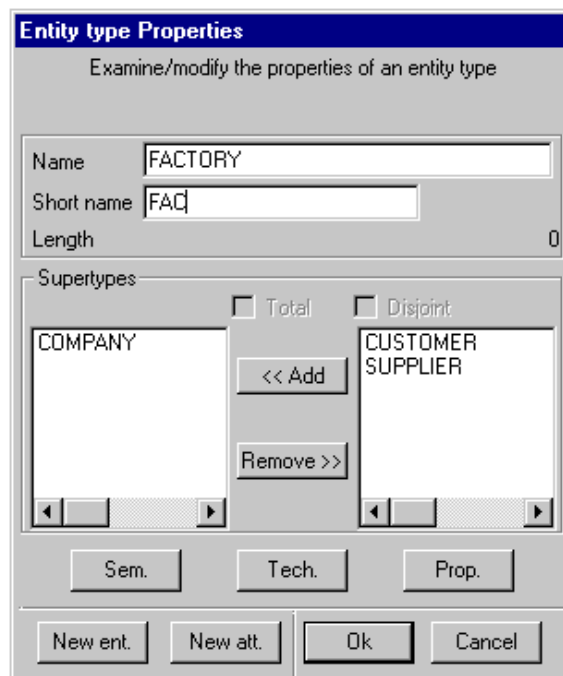


Figure 8.2 - FACTORY is being declared a subtype of COMPANY.

Defining similarly that SUPPLIER and CUSTOMER both have COMPANY as their supertype leads to the schema of Figure 8.3.

It is common to talk about **IS-A relation** between the supertype and its subtypes. The origin of this name lies in the natural language interpretation of the facts modeled in this way:

each supplier is a company, each factory is a company, etc.

The standard view is shown in Figure 8.4 and the extended view in Figure 8.5.

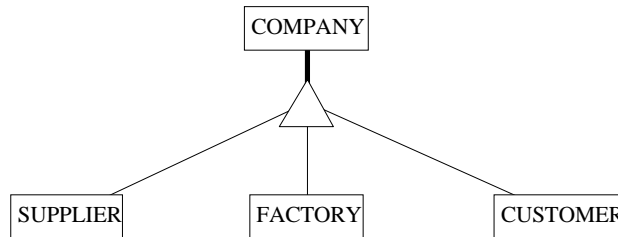


Figure 8.3 - SUPPLIER, FACTORY and CUSTOMER are subtypes of COMPANY



Figure 8.4 - The Text standard view of IS-A relations.

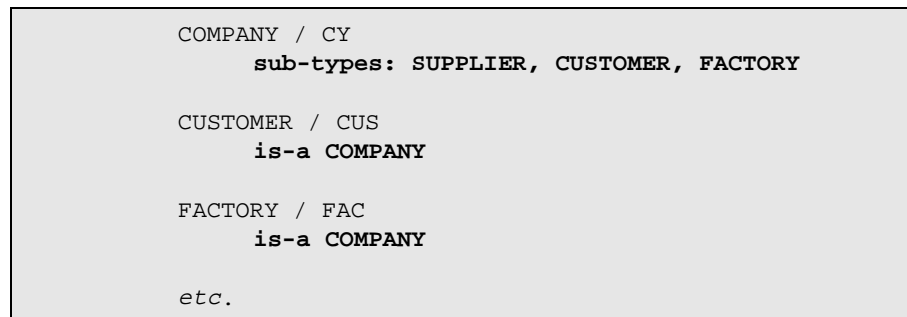


Figure 8.5 - The Text extended view of IS-A relations.

As we can guess by playing with the Entity box, it is possible to declare that an entity type has more than one supertype. However, such situations, often called *multiple inheritance*, are a bit more complicated. They will be discussed specifically later in this lesson.

8.3 Properties of the subtypes of an Entity type

So far, we have defined the relation between each subtype and its supertype: each entity of the subtype is an entity of the supertype. So we know that each customer is also a company, and so forth for factories and suppliers.

Now, what about a customer being a supplier as well? ... and about a company which is neither a customer, a factory, nor a supplier?

These questions address two main properties that concern the entity types involved into a supertype/subtype relation. The questions can be stated more formally:

- *are any two subtypes disjoint, or can they overlap*¹? If the subtypes are pairwise disjoint, then any supertype entity belong to at most one of its subtypes; otherwise it can belong to several subtypes. To assert this property, we will say that the subtypes of entity type COMPANY are **Disjoint**. Since this property concerns all the subtypes of COMPANY, it is considered to be a property of the supertype.
- *must each entity of the supertype belong to a subtype, or can it be in none of them?* If each supertype entity must belong to at least one subtype, we will say that the subtypes of entity type COMPANY are **Total**. This too is a property of the supertype.

When the collection of the subtypes of E is both disjoint and total, this collection forms a **Partition**. In a partition, each E entity belongs to exactly one subtype.

To allow us to declare these properties, the *Entity box* of the supertype includes two buttons, named **Disjoint** and **Total** (Figure 8.6). Each can be checked and unchecked independently. When both are checked, the subtypes form a **Partition**, that is, each COMPANY entity is of *exactly one* subtype.

1. To be more precise, this question concerns the set of entities of each type.

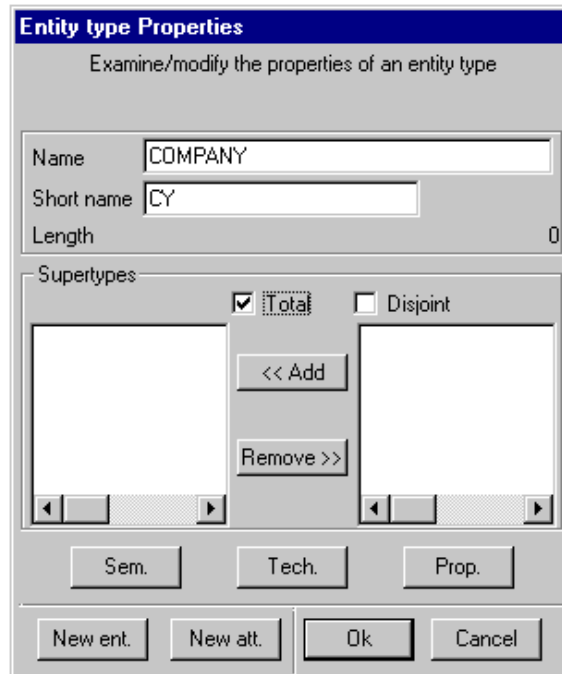


Figure 8.6 - The subtypes of COMPANY **totally** cover the entity set of COMPANY.

To practice these concepts, we define the subtypes of COMPANY as being **total**:

- we open the *Entity box* of COMPANY (by double-clicking on its name);
- we click on Total;
- we click on OK.

The schema appears as in Figure 8.7 and Figure 8.8.

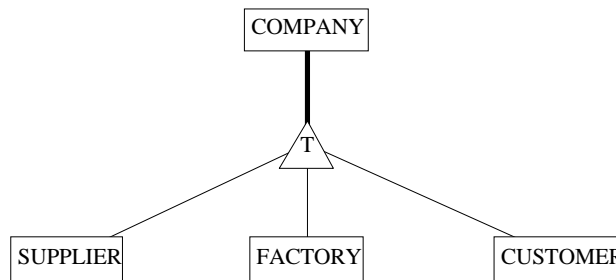


Figure 8.7 - Each COMPANY entity also is a SUPPLIER, a FACTORY or a CUSTOMER entity (or several of them).

```

COMPANY / CY
      sub-types (T): SUPPLIER, CUSTOMER, FACTORY

CUSTOMER / CUS
      is-a COMPANY

etc.
    
```

Figure 8.8 - The Text extended view of the IS-A relations of Figure 8.7.

The triangle symbol represents a collection of subtypes. This symbol can include an additional character: **T** for Total, **D** for Disjoint and **P** for Partition. The absence of character means both non-disjoint and non-total, i.e., an overlapping and partial collection of subtypes.

This point being very important in modeling, we will synthesize the different situations in Figure 8.9. It shows a simple IS-A hierarchy made up of super-type A and subtypes B1 and B2. Each pattern is defined as follows.

Total Disjoint

Partition: **each** A entity is either a B1 entity or a B2 entity **but not both**.

Total Disjoint

Total: **each** A entity is either a B1 entity or a B2 entity **or both**.

Total Disjoint

Disjoint: an A entity can be a B1 entity or a B2 entity **but not both**. Some A entities are neither B1 nor B2 entities.

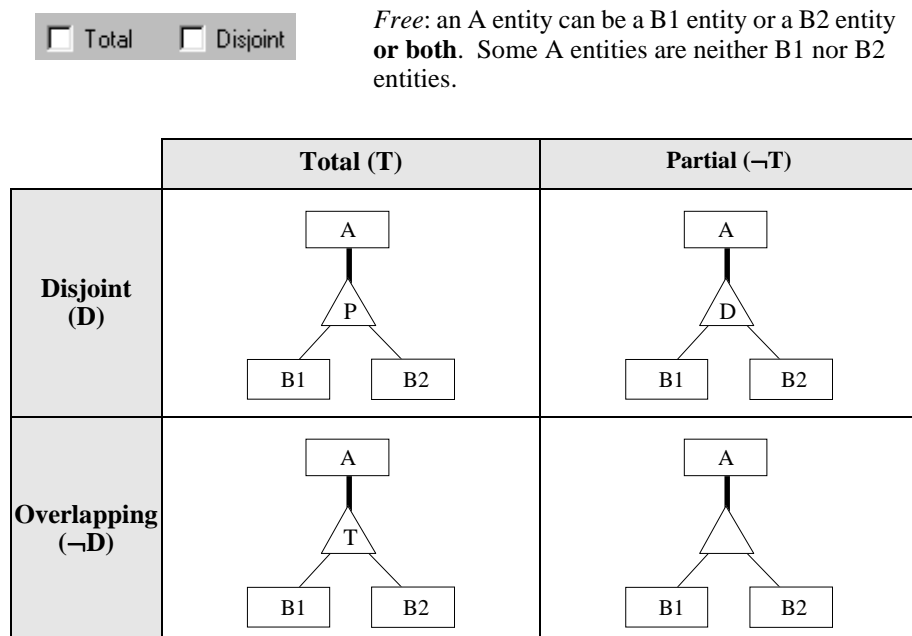


Figure 8.9 - Synthesis of subtype properties.

8.4 Supertype/Subtype inheritance

The Supertype/subtype IS-A relation is not as simple as it appears at first glance. One of its most dramatic consequences is the so-called **inheritance** mechanism. To describe it, we need first to enrich our schema a little bit by giving entity types some attributes. Let us record the following facts:

- each company has a name (identifier) and an address;
- each supplier has an account number;
- each factory has a production type;
- each customer has a customer number (identifier), a status and an amount due.

The current schema can be completed easily (Figure 8.10).

Though it is quite correct, this schema does not show explicitly all its contents. For instance, each *customer*, being a *company*, has also a *name* (which identifies it) and an *address*.

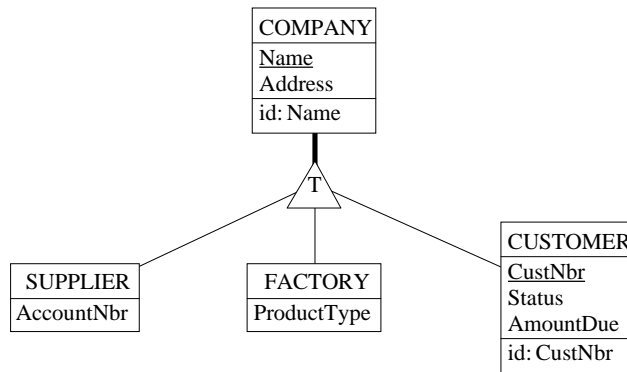


Figure 8.10 - An IS-A hierarchy with attributes.

Thus, the whole list of attributes of entity type CUSTOMER is in fact made of: CustNbr, Name, Address, Status and AmountDue. Among them, CustNbr, Status and AmountDue are called the **proper attributes**, while Name and Address are the **inherited attributes**. In addition, CUSTOMER has two identifiers, namely CustNbr (a proper identifier) and Name (an inherited identifier).

Should the schema show all the attributes and all the identifiers of each entity type, it would appear as in Figure 8.11.

The first version is more concise, while the latter is more informative and includes redundant specifications². However, both views have the same information contents. The only difference is how we have to interpret them.

The concept of inheritance also applies to all the structural properties of the entity types, and is not restricted to attributes and identifiers as discussed so far. More specifically, the subtypes also inherit all the *roles* and the *integrity constraints* of their supertype.

For instance, if COMPANY is linked to entity type REGION, then CUSTOMER, FACTORY and SUPPLIER are linked to REGION as well (Figure 8.12). Its explicit semantic contents are shown in Figure 8.13.

2. For instance, it tells us *twice* that a *customer has a name*: once through an inherited attribute and once as a proper attribute of the supertype.

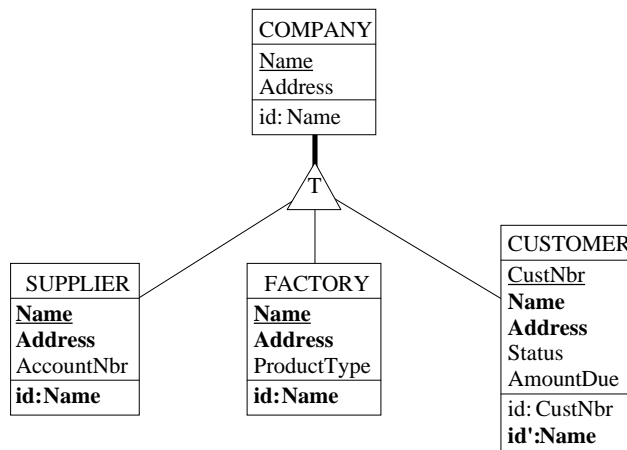


Figure 8.11 - Attribute and identifier inheritance explicitly shown. The inherited components are marked for readability.

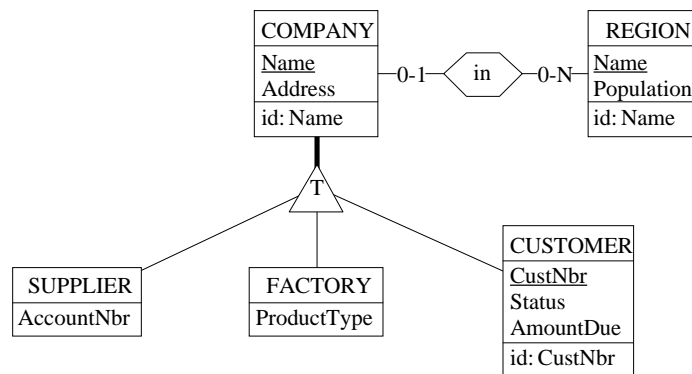


Figure 8.12 - The supertype plays a role in a rel-type.

By comparing both views, the gain of conciseness induced by the supertype/subtype relation is obvious, specially in large schemas. There are other advantages as well. For instance, inherited components are described only once at the supertype level. Therefore, changing the definition of an attribute (or a role), adding an attribute or deleting an existing attribute, must be done only on-

ce. All these changes are automatically applicable to all the subtypes of the supertype.

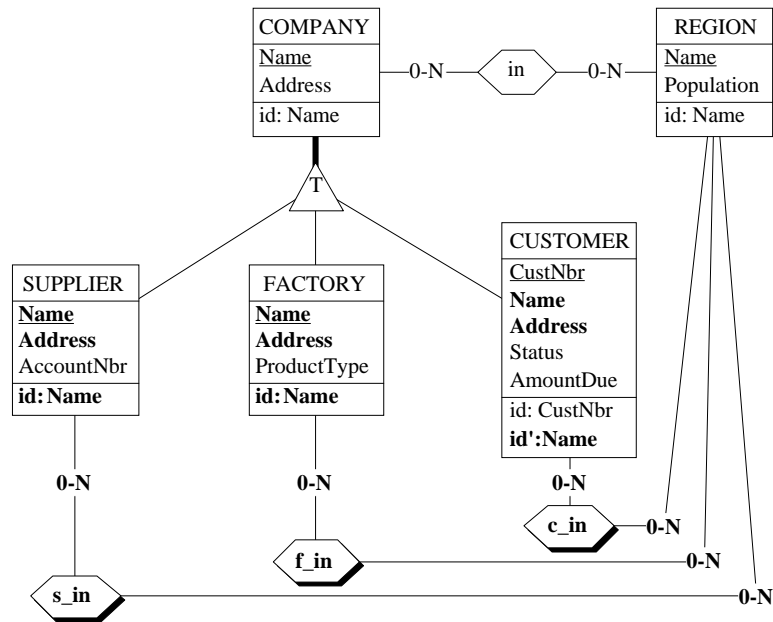


Figure 8.13 - Attribute, identifier and role inheritance shown explicitly.

The drawback of IS-A constructs is that the schema can be less readable. Indeed, the actual attributes (and other components) comprise the proper attributes + the inherited ones.

8.5 Multilevel IS-A hierarchy

The example developed in this lesson includes one level of subtypes only, forming a 2-level hierarchy. Some problems require deeper hierarchies, as illustrated in Figure 8.14. This schema classifies the documents available in a corporate library. The rules discussed above still are valid for more than 2 levels. For instance, a *scientific book* is a *book*, which in turn is a *document*. Therefore, any *scientific book* is a *document* as well, and thus inherits from both *books* and *documents*.

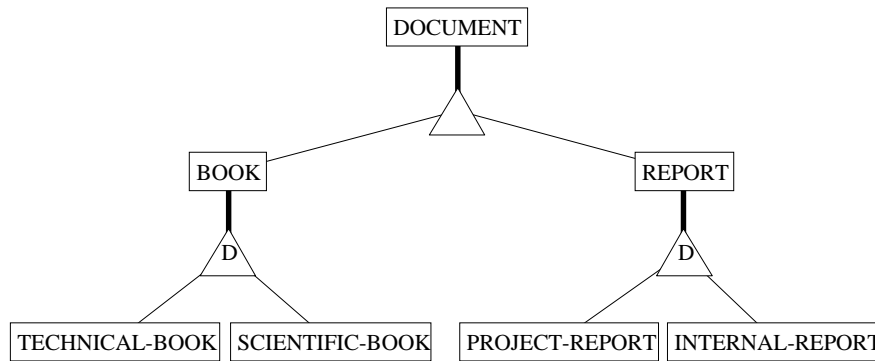


Figure 8.14 - A 3-level IS-A hierarchy.

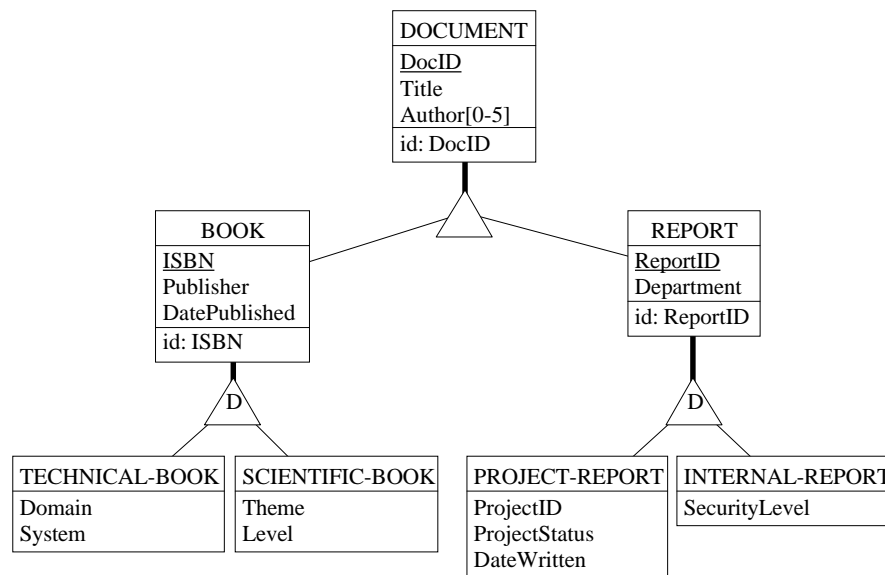


Figure 8.15 - The IS-A hierarchy with all the proper attributes and identifiers shown.

Let us assign some plausible attributes to each of these entity types (Figure 8.15). According to the inheritance rules, each subtype is assigned the attribu-

tes (and constraints and roles) of all its direct or indirect supertypes (Figure 8.16).

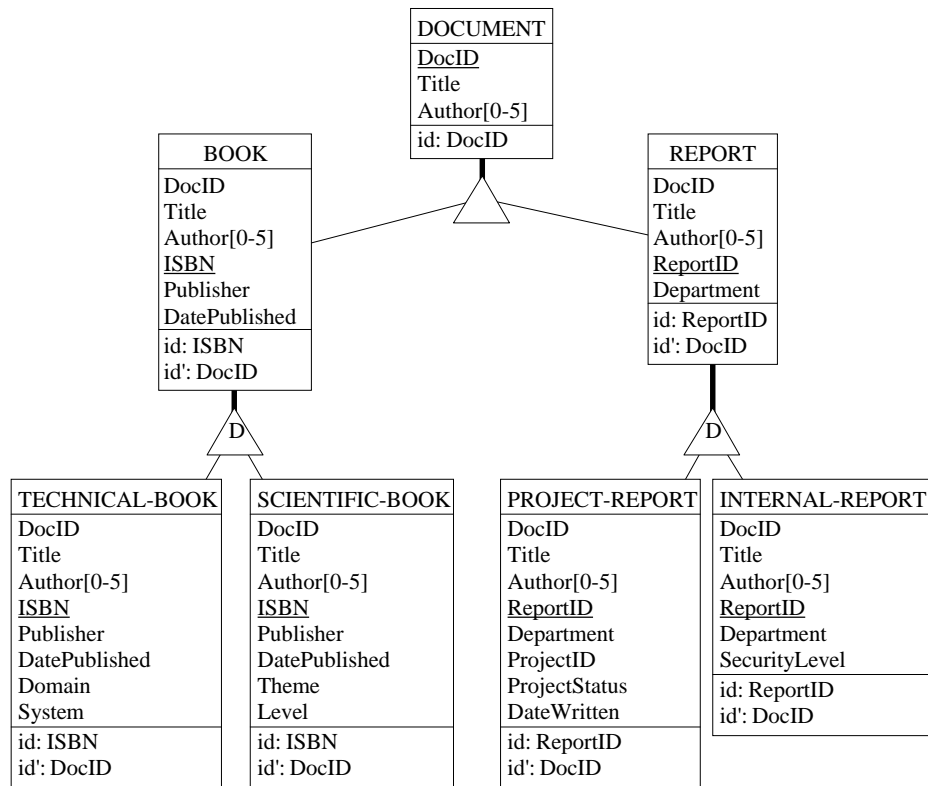


Figure 8.16 - The IS-A hierarchy with all the proper **and inherited** attributes and identifiers shown.

8.6 Multiple inheritance

So far, any entity type has at most one supertype from which it inherits a part of its properties. Situations may occur that require a subtype to inherit from more than one supertype. Considering the schema of Figure 8.15, we can imagine some *reports*, for instance *project reports*, being published as *scientific books*. So, these *published reports* are both *project reports* and *scientific books*. As a consequence, a published report has *report ID* and a *project status*

(as project report) as well as an *ISBN* and a *theme* (as scientific book). Such a structure, illustrated in Figure 8.17, generally is called **multiple inheritance**. The term is a bit improper, since inheritance is just a consequence of the IS-A relation. A better name would be **multiple IS-A hierarchy**.

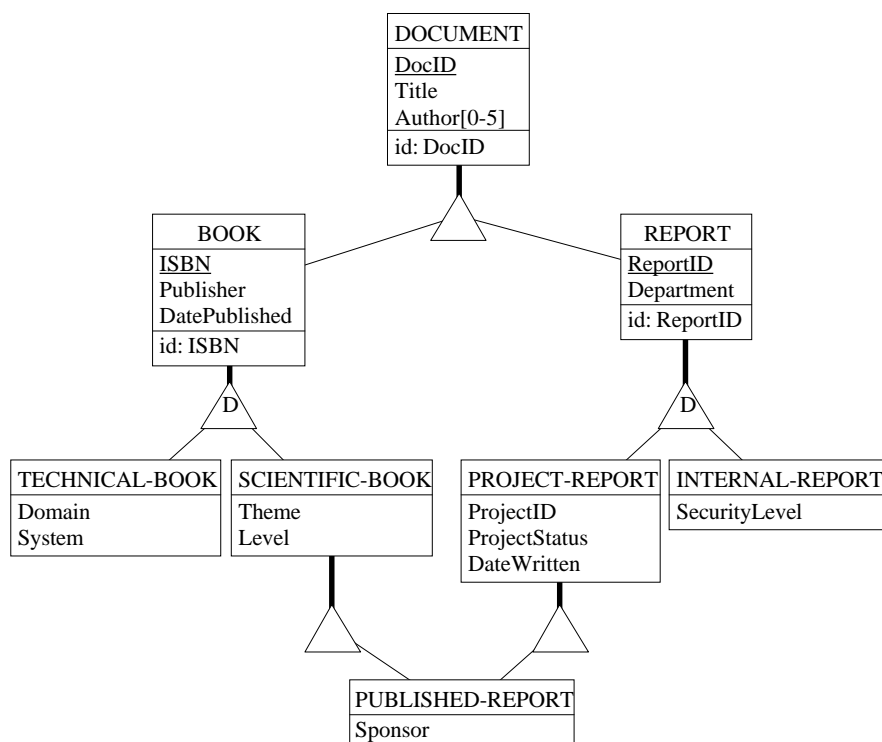


Figure 8.17 - Example of multiple inheritance: PUBLISHED-REPORT is both a SCIENTIFIC-BOOK and a PROJECT-REPORT.

This example raises interesting questions.

1. Let us first observe that PUBLISHED-REPORT has a single ancestor, namely DOCUMENT, which can be found by navigating upwards through the left branch (SCIENTIFIC-BOOK → BOOK → DOCUMENT) or through the right branch (PROJECT-REPORT → REPORT → DOCUMENT).
2. Now, let us consider the subtypes of DOCUMENT. They have been declared *overlapping* (no D nor P symbols), so that some documents can be

both books and reports. It is fortunate, because a published report being both a book and a report, this overlapping property makes it possible to have published reports. Think of what would have happened regarding PUBLISHED-REPORT if the subtypes of DOCUMENT were declared disjoint³!

- Note that PUBLISHED-REPORT is **not the intersection** of SCIENTIFIC-BOOK and PROJECT-REPORT⁴, but more generally is a **subset of this intersection**.

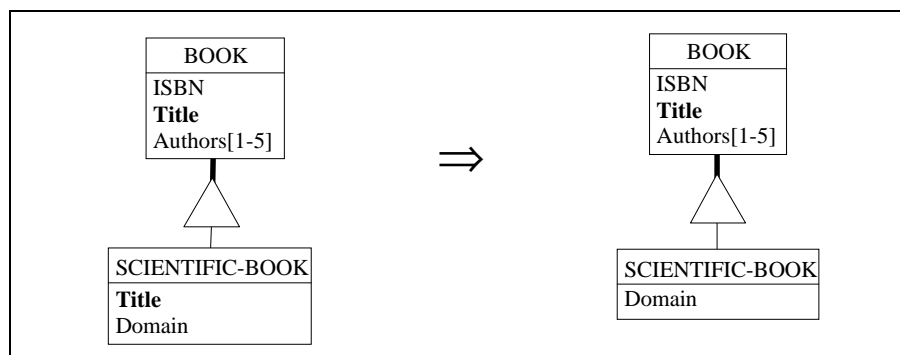


Figure 8.18 - The supertype and the subtype have an attribute with the same name. If they represent the same meaning, one of them only must be kept.

Let us now discuss the inherited attributes. Three conflicting problems may arise and must be addressed.

- What if a subtype has a proper attribute with the same name as some attribute of the supertype (Figure 8.18)? If these attributes are the same and describe the same real world property, one of them must be removed. Otherwise they have different meanings and the name of one of them must be changed. Indeed, the attributes of the supertype are attributes of the subtype, and in any entity type, the attributes of the same level must have distinct names⁵.

-
- This would be an interesting example of **inconsistent structure**, i.e., a structure no data will ever satisfy. Indeed PUBLISHED-REPORT would always be empty!
 - As already mentioned, we identify an *entity type* and its *population* at any given time. Not quite correct but very handy to simplify the discussion.

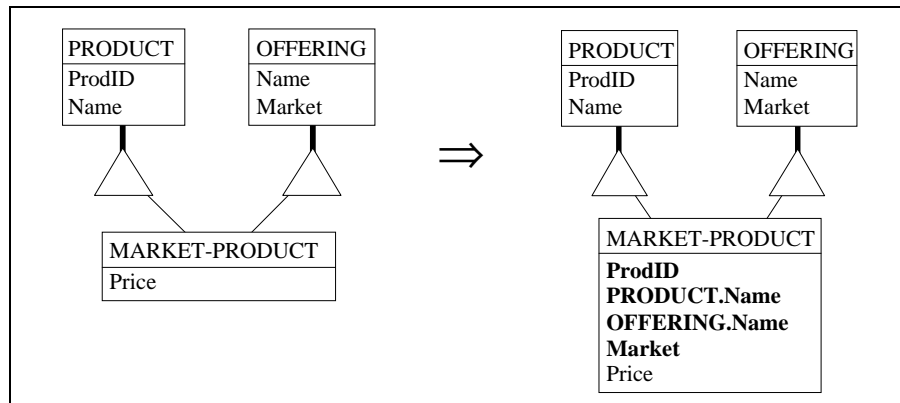


Figure 8.19 - A *market product* is a *product* that is offered to consumers. Therefore, it has an *internal product name* and an *offering name*, that can be different. When considering these names from the MARKET-PRODUCT viewpoint, it is good practice, to solve the naming conflict, to think of them as being prefixed with the name of their source entity type.

2. Two independent entity types can each have an attribute with the same name. What if they share a common subtype? The latter inherits two different attributes with the same name. A good practice can be to prefix the inherited attributes with the name (or unique short name) of their source entity type (Figure 8.19).
3. In a multiple IS-A hierarchy, an entity type inherits from one or several common ancestors through more than one branch. Therefore, the ancestor's attributes are inherited more than once! Of course, for each of them, only one must be considered (Figure 8.20).

Now, considering the schema of Figure 8.17, we can state exactly what the attributes and identifiers of entity type PUBLISHED-REPORT are (Figure 8.21).

-
5. In some models, the designer is allowed to change the definition of an inherited attribute. For instance, its domain of values can be restricted to a subset of that of the origin attribute. To simplify the discussion, we will ignore this possibility.

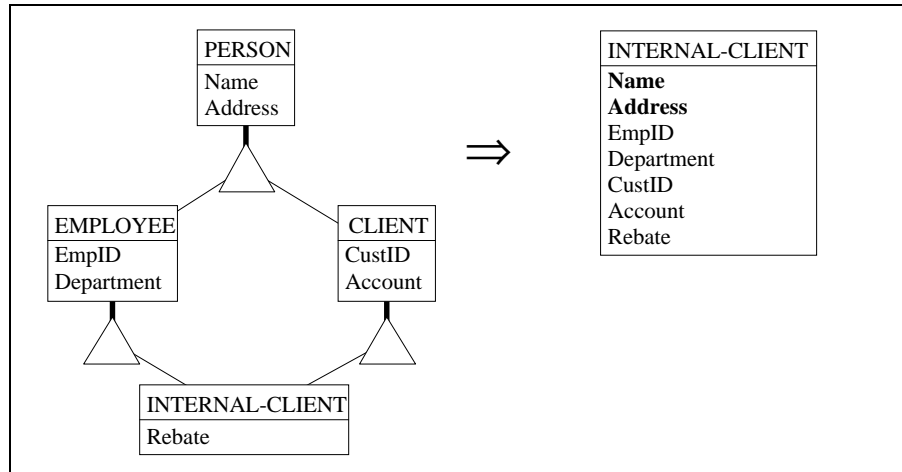


Figure 8.20 - INTERNAL-CLIENT inherits from PERSON twice. However, the attributes of the ancestor are inherited only once.

PUBLISHED-REPORT
DocID
Title
Author[0-5]
<u>ISBN</u>
Publisher
DatePublished
Theme
Level
ReportID
Department
ProjectID
ProjectStatus
DateWritten
Sponsor
id: ISBN
id': DocID
id': ReportID

Figure 8.21 - The proper and inherited attributes and identifiers of entity type PUBLISHED-REPORT from Figure 8.17.

8.7 Processing units of a schema

An entity type represents the existence and the properties (attributes and constraints) of a class of outstanding objects of the application domain. Besides this static view, we could want to describe the behaviour of these objects. The standard way is through **operations** or **methods**, as proposed in object-oriented approaches.

A method is a service associated with an entity type. Each entity⁶ of this type can respond to any call for this service (this *call* is generally called a **message**).

Figure 8.22 shows some methods associated with entity types CUSTOMER and ORDER in the form of **processing units**.

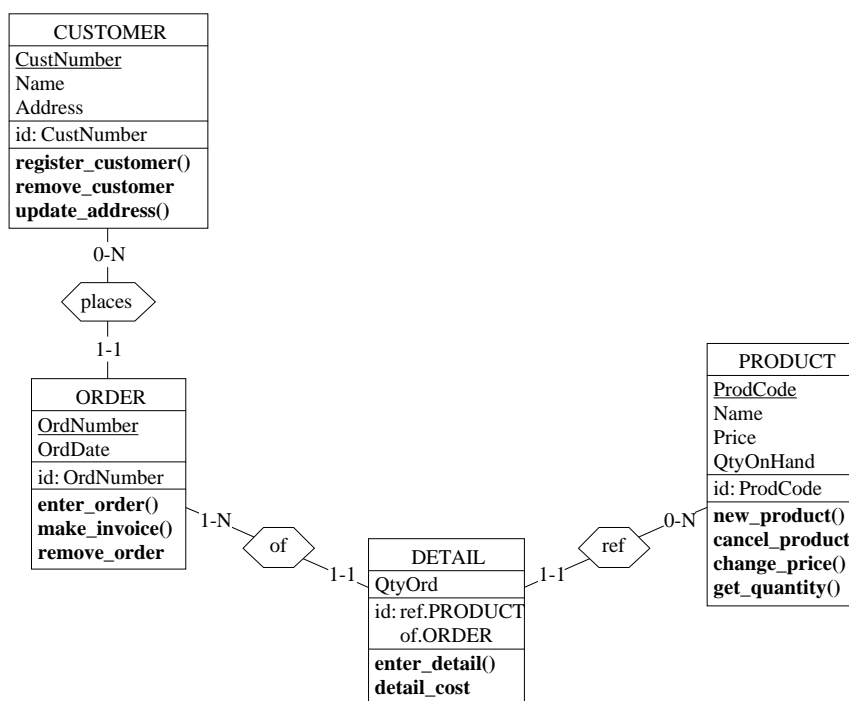


Figure 8.22 - Each entity type has been given processing units, or **methods**.

- The OO approach distinguishes *class methods*, which the entity type is responsible for, from *instance methods*, that can be taken in charge by the entities. For instance, `register_customer` is a class method that must be asked to entity type CUSTOMER while `remove_customer` is an instance method.

A processing unit is defined by selecting the entity type and by executing the command **New / Processing unit** (Figure 8.23).



The dialog box titled "Processing Unit Properties" is used to create a processing unit for the entity type "PRODUCT". It contains the following fields and buttons:

- Title: "Create processing unit of PRODUCT"
- Name field: "new_product()"
- Short name field: (empty)
- Buttons: "Sem.", "Tech.", "Prop.", "Next proc.", "Ok", "Cancel"

Figure 8.23 - Defining a processing unit for entity type `PRODUCT`.

Processing units can be associated with rel-types and schemas (Figure 8.24) as well. In the latter case, they represent global functions of the system, such as organizational functions in a conceptual schema or application programs in a logical schema.

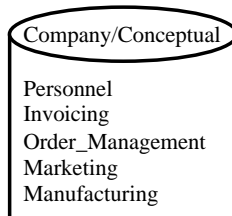


Figure 8.24 - Global procedures associated with a schema.

Processing units can be inherited too. However, special mechanisms will be used, such as **overloading**. More on this later on.

Key ideas of Lesson 8

1. Supertypes and subtypes

An entity type can be a subtype of one or several other entity types, called its supertypes. If B is a subtype of A, then each B entity is an A entity as well. The collection of the subtypes of entity type A is declared total (symbol T) if each A entity belongs to at least one of its subtypes; otherwise, it is said to be partial. This collection is declared disjoint (symbol D) if an entity of a subtype cannot belong to another subtype of B; otherwise, it is said to overlap. If this collection is both total and disjoint, it forms a partition (symbol P).

Since a supertype/subtype relation is interpreted as *each B entity is an A entity*, it is called an IS-A relation. IS-A relations form what is called an IS-A hierarchy.

An entity type can have more than one supertype. Such a situation is called multiple IS-A hierarchy, or more commonly (but improperly) multiple inheritance.

2. Inheritance

Since all B entities are A entities as well, entity type B inherits all the properties of entity type A. In particular, all the attributes of A are attributes of B as well. This is true too for the identifiers and other constraints, as well as for all the rel-types in which A participates.

In a multiple IS-A hierarchy, some rules must be satisfied in order to make the inheritance mechanism conflict-free.

3. Processing units

Procedures can be associated with entity types, rel-types and schemas, to represent the behaviour of the system described by the schema. Processing units of entity types can be used to define methods in object-oriented schemas.

Summary of Lesson 8

- In this lesson, we have studied the following notions:
 - supertypes, subtypes, supertype/subtype relation or IS-A relation
 - total, disjoint and partition properties of the collection of the subtypes
 - inheritance of attributes, constraints and rel-types
 - processing units associated with entity types, rel-types and schemas

- We have also learned
 - to specify the supertype(s) of an entity type:
 - in the Entity type box of the subtype, include the name(s) of the supertype(s) in the Supertype list box
 - to define the total, disjoint properties:
 - in the Entity type box of the supertype: click on the buttons Total, Disjoint.
 - to define a processing unit:
 - New / Processing unit.**

Exercises for Lesson 8

- 8.1 In the beginning of this lesson, we wrote: ... *factories, suppliers and customers are companies*. In addition, *each factory can have customers and can have suppliers*. ...

Complete the corresponding schema in order to include these specifications.

- 8.2 In the same schema, describe the fact that each company can be *a subsidiary of another company* (Hint: use a cyclic relationship type). Show how this fact must be interpreted through the IS-A relation. In other words, make explicit the inherited relationship type. On the basis of this small example, what do you think of the conciseness of IS-A relations?

- 8.3 Build a schema (called PERSONNEL) representing the following application domain.

The company has employees. Each of them is identified by an employee id, and has a name and an address. An employee can have a personal file. This file has an identifying code, a date and a content. Among the employees, there are clerks and workers. Workers are characterized by a salary, and must be affiliated to a trade union. A clerk has a level and a function. A trade union has a name and an address.

Consider four different hypotheses:

- each employee is either a clerk or a worker, but not both (version 1);
- an employee can be a clerk or a worker, but not both (version 2);
- each employee is either a clerk or a worker, or both (version 3);
- an employee can be a clerk or a worker, or both (version 4).

- 8.4 Derive from one of these schemas another schema which makes all the properties of each entity type explicit by showing the effect of the inheritance mechanism.

- 8.5 An application domain concerns vehicles. Some are cars while others are trucks. There are special vehicles that are both cars and trucks. Imagine two or three different kinds of cars and two kinds of trucks. Draw the Entity-relationship schema of this application domain and define proper attributes for each of these entity types.

Lesson 9

More about attributes

Objective

Attributes form the building blocks of every schema. In this lesson, we examine new forms and new features of attributes. We learn more on domains (particular user-defined domains), multi-valued attributes, stable and non-recyclable attributes, attribute identifiers, multivalued identifiers, reference attributes and access keys and object attributes.

Some of these concepts are specific to conceptual schemas, while others will be used in logical schemas.

9.1 Introduction

If entity types are the building block of schemas, attributes are those of entity types. Hence the need for a rich set of features to define the many kinds of attributes that appear in all the data models in which databases can be specified.

9.2 Built-in domains

Built-in domains are very general, meaningless, data types that are proposed by most programming languages, DBMSs and CASE tools. *Character strings*, *numeric* and *date* data types are some of the most common built-in domains. The DB-MAIN CASE tool proposes eight of them through the Type list-box of the Attribute property box (Figure 9.1).

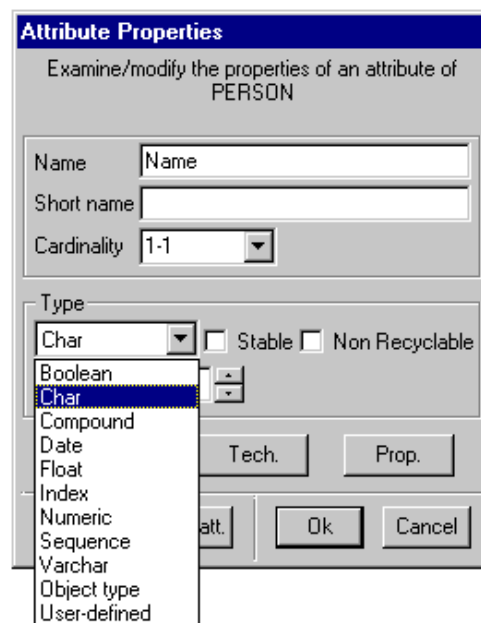


Figure 9.1 - Atomic attributes can be based on built-in domains: *boolean*, *Char(acter)*, *Date*, *Float*, *Index*, *Numeric*, *Sequence* and *Varchar(acter)*. An attribute is *Compound* if it is given component attributes.

1. *Boolean*(*n*): Set {true, false}, or any set of 2 elements.
2. *Char*(*n*): The set of **n**-character strings.
3. *Varchar*(*n*): The set of strings with length from 0 to **n**.
4. *Date*(*n*): Set of dates or timestamps.
5. *Numeric*(*n*,*d*): Set of numerical values of **n** decimal digits, including **d** decimals.
6. *Float*(*n*): Set of floating point numerical values with a representation of **n** bytes.
7. *Index*(*n*): Numerical values that designate the elements of parent attribute A[I-J], which is a multivalued attribute of type *array*. If A has actual cardinality *k*, the index attribute instances takes **some** values from 1 to *k*. More on this in Section 9.6.
8. *Sequence*(*n*): Numerical values that designate the elements of parent attribute A, which is a multivalued attribute of type *list*. If A has actual cardinality *k*, the index attribute instances takes **all** the values from 1 to *k*. More on this in Section 9.6.

In these definitions, **n** stands for the length (*Boolean*, *Char*, *Date*, *Numeric*, *Float*, *Index*, *Sequence*), or the max length (*Varchar*), of the domain values. For each type, the tool proposes a default length. Except for *Date* and *Boolean*, it is an unusual value that should, in most cases, be replaced. Indeed, most *Char*(**1**) values must be considered as a length that the analyst forgot to set. The *Schema analysis assistant*¹ can easily detect this pattern.

The table of Figure 9.3 describes the rules for **n**. Figure 9.2 shows some examples of usage of built-in domains.

<pre> PRODUCT ProdNum: num (8) Name: char (28) Description: varchar (N) Price: num (8,2) Available: boolean id: ProdNum </pre>
--

Figure 9.2 - Some attributes based on built-in domains.

1. Will be described later on.

Type	range of n	default	particular rule
Boolean	1-99	1	
Char	1-99999	1	
Varchar	1-99999; N	1	N stands for <i>unlimited length</i>
Date	1-99	10	
Numeric	1-99; 0-99	1	1st figure = total length 2nd figure = decimals
Float	1-99	1	
Index	1-9	1	not shorter than length of max card. of the array (e.g., 3 for max card. = 500)
Sequence	1-9	1	not shorter than length of max card. of the list

Figure 9.3 - Rules for length *n* of each built-in domain: *range*, *default length* and some *particular rules*.

9.3 User-defined domains

Built-in domains convey almost no semantics: just numbers, character strings and the like. What a pity for such essential elements! What about defining our own, semantics-bearing, domains? That's the goal of *user-defined* or *application-specific* domains.

A user-defined domain is like an attribute, except that it has no parent, and can be used as a domain for true attributes. Let us consider an application domain in which several attributes are some variants of personal ID, addresses, phone numbers and VAT numbers. It would be most convenient if we could use special domains called `PID`, `Address`, `PhoneNumber` and `VATnumber` instead of meaningless built-in domains `numeric` and `character strings`.

Using user-defined domains has several advantages:

- *incremental modeling*: discovering the main information types of the application domain is the first step in several information system design methodologies; in any methodologies, user-defined domains can be considered as a first level of reusable components;

- *readability*: the semantics of attribute can be made more explicit, making the schema much more informative;
- *maintainability and ease of evolution* the name or the definition of a domain can be changed at one place, then propagated to all the attributes that use it; for instance, modifying the structure of phone numbers in all the schemas of a project can be made in a centralized way;
- *consistency*: attributes that denote similar properties and concepts can be based on the same domain, therefore increasing the simplicity and coherence of the specifications;
- *code generation*: an increasing number of DBMSs include domain declaration statements (`create domain` of SQL) or even abstract data types.

Needless to say that finding the optimal set of user-defined domains is an important asset when building successful and maintainable information systems.

Now, let us go in for practical aspects of user-defined domains.

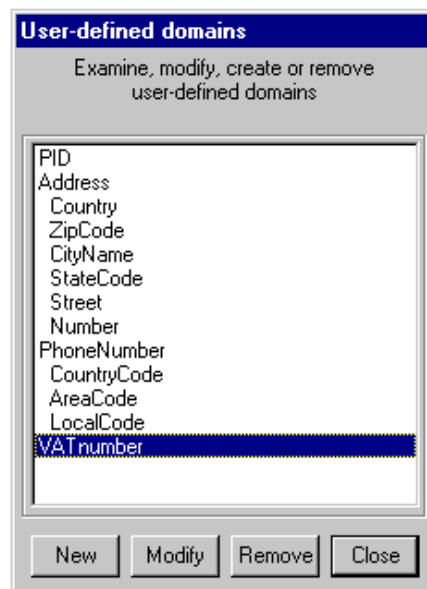


Figure 9.4 - The Management box of User-defined domains shows four domains that have been defined already, two of which being compound.

We call the User-defined domain management box through command **Product / User domains** (Figure 9.4). This box shows the domains already defined, and allows us to create new domains, to modify and to remove existing domains.

Defining a domain is just like defining a single-valued, mandatory attribute (cardinality [1-1]), and is done through the same Property box. A domain can be atomic or compound. A user-defined domain, or a component of it, can be based on built-in domains or on other user-defined domains. Take the necessary time to write a precise semantic description of each domain you define. As you can expect, recursive domains, i.e., domains defined directly or indirectly on themselves, are not allowed!

When defining an attribute, we can select a user-defined domain by choosing User-defined in the Type list box (Figure 9.1). Then, a new field appears (User-def.), showing all the available user-defined domains (Figure 9.5). Just select one of them. Do practice this concept by defining the domains of Figure 9.4 and by entering the schema of Figure 9.6.

Figure 9.5 - Any attribute can be defined on a User-defined domain.

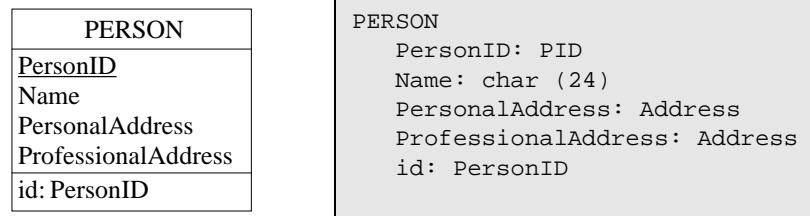


Figure 9.6 - PersonID has been defined on user-defined domain PID, while PersonalAddress and ProfessionalAddress have been defined on domain Address.

9.4 Stable and non-recyclable attributes

Standard attributes have values that represent definite states of the application domain at a given time: the address of an employee, the account level of a customer or the description of a product. As a consequence, attribute values of an entity evolve according to the changes of state of this entity: employees move, account levels go up and down and a product can get another description.

However, we can want some properties of an entity type to be stable during the life of each entity. Therefore, allowing users to change the values of the corresponding attributes can be considered undesirable.

Attributes can be given two special properties (Figure 9.7):

- **stability**: once an entity has received a value for this attribute, this value can never be changed in the life of the entity;
- **non-recyclability**: once an entity has received a value for this attribute, this value can never be reused for another entity, even long after the first entity was deleted².

Of course, most attributes are *unstable* and have *recyclable* domains.

These properties are closely linked with the identifiability of entities, and mainly concerns primary identifiers. The best application of the concept surely is *history representation and management*. Let us consider that we want to record all the successive states of persons initially described as in Figure 9.6.

2. In fact, the recyclability property is a characteristic of the domain of the attribute.

Attribute Properties
Examine/modify the properties of an attribute of
PERSON

Name: PersonID
Short name:
Cardinality: 1-1

Type
User-defined Stable Non Recyclable
User-def.: PID

Sem. Tech. Prop.

First att. Next att. Ok Cancel

Figure 9.7 - PersonID is **stable**, that is, once assigned to an entity, its value cannot be changed, and **non recyclable**, so that no assigned values can ever be reused, even when the entities have been deleted.

Now each state of each person is represented by a HIST_PERSON entity (Figure 9.8), in which Begin and End are timestamp values that define the period during which the person was in this state. When any property (name, address, etc.) of a person changes, the current state is closed and a new current state is created. A state is identified by the PID of the person and the Begin time of the state. The complete history of the person with PID = X is represented by the sequence of all HIST_PERSON entities that have PID = X.

What would happen if attribute PID value were allowed to change during the life of a person? Clearly, it would be impossible to rebuild the history of this person, because we would have lost the only common property of the states, that is the unique PID value. *Conclusion*: PID must be declared **stable**.

Now, we suppose that we keep the histories of all the entities that have been deleted (*deletion* is the ultimate *state change*). It is quite obvious too that once a PID value has been assigned to an entity, be it currently alive or deleted, it can never be assigned to another entity, otherwise the concept of history would become ambiguous. Indeed, all the histories with the same PID value, though

they come from different persons, would coalesce into a single history. Therefore, PID must be declared, not only **stable**, but also **non recyclable**.

HIST_PERSON
<u>PersonID</u>
<u>Begin</u>
End
Name
PersonalAddress
ProfessionalAddress
id: PersonID
Begin

Figure 9.8 - Representing *histories* of persons.

9.5 Attribute identifiers

So far, entity types and rel-types can be given identifiers. On the other hand, each value of a multivalued attribute is a set of values, i.e., a collection of distinct values. In the schema of Figure 9.9, all `Sales` values are unique, that is, no two `Sales` values are made up of the same `Region` value and the same `Year` value and the same `Volume` value.

SALESMAN
<u>PID</u>
Name
Address
Sales[0-20]
Region
Year
Volume
id: PID

Figure 9.9 - Each `SALESMAN` entity has a set of distinct `Sales` values.

However, we feel that the `Volume` component is useless to state the uniqueness property of the `Sales` values. Clearly, the values of `Region` and `Year` should suffice to identify one `Sales` value among all those of a given `SALESMAN` entity. One way to assert this property is to say that the values of `Sales`, for any given `SALESMAN`, are identified by `{Region,Year}`. In

other words, {Region,Year} is the **identifier** of multivalued attribute Sales of SALESMAN.

We define such an identifier as follows.

1. We select parent attribute Sales.
2. We execute command **New / Group**. This opens the Group property box for Sales.
3. We move components Region and Year in the Component field.
4. We click on the Primary ID button, then on OK.

The resulting schema should look like that of Figure 9.10. Note that more than one identifier can be declared for an attribute.

SALESMAN
<u>PID</u> Name Address Sales[0-20] Region Year Volume
id: PID id(Sales): Region Year

Figure 9.10 - For any particular salesman, we record yearly sales for each region. Therefore, the Sales values of each SALESMAN entity have distinct Region and Year values. {Region,Year} is declared the primary identifier of attribute Sales.

To make the concept more understandable, it can be useful to give an equivalent form of this schema. Let us suppose that Sales are represented as SALES entities instead. Rel-type `for` links SALESMAN with SALES. What about the identifier of SALES? Since *a salesman cannot make two sales in the same region, the same year*, the identifier must comprise {`for`. SALESMAN, Region, Year} as shown in Figure 9.11. So, the schemas of Figure 9.10 and Figure 9.11 convey exactly the same semantics.

There is a nice trick to show this, namely a *schema transformation*. First open the schema of Figure 9.10, then proceed as follows.

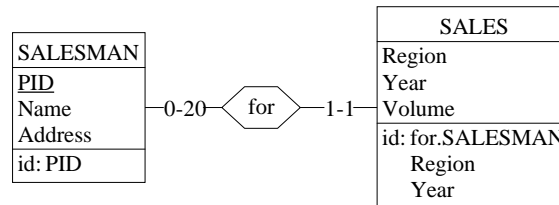


Figure 9.11 - The right side schema is another equivalent way to represent the same situation.

1. We select attribute `Sales`.
2. We execute command **Transform / Attribute / -> Entity type**. A small box opens (Figure 9.12), which we validate.
3. We choose the name of the new entity type (`SALES`) and of the new rel-type (`for`).

Surprise! We get the schema of Figure 9.11. If you are not fully convinced, try the inverse transformation: select entity type `SALES`, then execute **Transform / Entity type / -> Attribute**.

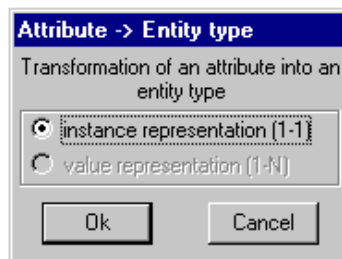


Figure 9.12 - This box asks us what kind of *attribute-to-entity-type* transformation we want. So far, we have not the slightest idea of what this could mean! Nevertheless, considering the limited list of possible techniques, it should not be too difficult to choose.

9.6 Non-set multivalued attributes

Several conceptual models (ERA, Merise, NIAM, etc.) include the concept of multivalued attribute, allowing analysts to model associations of sets of values with entities, while other models, such as OMT and UML, do not propose it as standard. Though this construct is simple and (generally) well defined, it shows its weakness when more complex situations are to be modeled. Ordered sets, collections of non-distinct elements and arrays are among the most frequent non-set structures that appear to be difficult to specify and to understand. First, we will try understand why non-set collections of values can be useful. Then, we will propose an overview of these *set* and *non-set* structures, describing the main aspects of multivalued attributes and their conversion into pure set-oriented constructs.

As a first example, let us consider the phone numbers of a population of persons. We observe that each person can have from 0 to 5 phone numbers, which we translate in the schema of Figure 9.13.

PERSON
<u>PersonID</u>
Name
PersonalAddress
ProfessionalAddress
Phone[0-5]
id: PersonID

Figure 9.13 - Persons have from 0 to 5 phone numbers.

However, things are a bit more complex. Indeed, the phone numbers of a person are not of equal importance. The first number is the the main one to try, then, when it fails, we try the second one, and so forth to the last one. In other words, phone numbers form an *ordered list of 0 to 5 distinct values*. This is easy to specify, as shown in Figure 9.14. The result appears in Figure 9.15, where u-list stands for Unique list.



Figure 9.14 - Multivalued attributes can define mere sets. However, they can alternately form bags, lists, lists of unique values, arrays or arrays of distinct values.

PERSON
<u>PersonID</u>
Name
PersonalAddress
ProfessionalAddress
Phone[0-5] u-list
id: PersonID

Figure 9.15 - Multivalued attribute Phone is declared a *Unique list*: for each entity, its value form a sequence of unique values.

Now, it is time to examine all the collection types provided by the DB-MAIN model. For each of them, we propose an intuitive definition based on a practical example. Afterwards, we will show how non-set constructs can expressed as pure set equivalent expression. The presentation is a bit tedious, but it is worth being carefully followed.

Sets

An attribute A depending on parent object P is called multivalued if a collection of more than one value of A can be associated with each instance of P. Otherwise, the attribute is called single-valued. The cardinality of the attributes states of how many values this collection can be made up. The schema of Figure 9.16 (left) partly models an application domain in which each customer can be given from 1 to 5 phone numbers.



Figure 9.16 - Phone is a **set** multivalued attribute of its parent CUSTOMER.

In the most simple situations, this value collection is just a pure set, which means that the values are distinct, and that no ordering (or whatever else) relation holds among the values. For instance, if a customer happens to have 4 phone numbers, its entity will be given a collection of four distinct Phone values in which no number can be considered as the first or the last one. Venn diagrams are commonly used to graphically represent such collections of values (Figure 9.16 - right). Except when specified otherwise, multivalued attributes are sets.

Bags

This is nice for phone numbers (and for some other interesting situations), but what about the following problem: our customer, besides having phone numbers, usually has cars too, for which we only want to record the make of each of them. Since people tend to buy cars of the same make, we must accept to record the same make name more than once. Therefore, we will associate, with each CUSTOMER entity, a collection of make names in which certain names can appear more than once. This particular form of collection in which different elements can be identical is called a **bag**³ of values. Figure 9.17 shows entity type CUSTOMER as well as the graphical representation of a bag

3. ... or **multiset**.

of Car values. In this example (right), the customer has one Ford car, one Toyota and two VWs.



Figure 9.17 - Car is a **bag** multivalued attribute.

Unique lists

For some particular kinds of information, the order of the values in each collection is meaningful. It is the case for the christian names of persons, among which there a first one, usually used to call this person to mow the lawn on saturday morning. Then there is a second one, generally reduced to its initial. Most often there is a third one, and some persons can have a fourth one. So the christian names of a person are distinct, but they are ordered. Such a collection will be called a **unique list**. Figure 9.18 illustrates this situation and a sample collection of christian names.

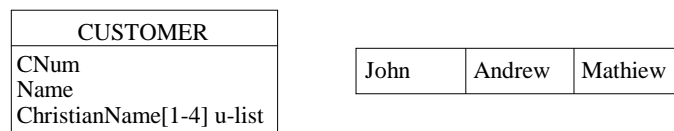


Figure 9.18 - ChristianName is a **unique list** multivalued attribute.

Lists

Of course, we could ask for the same privilege as we did for sets: being allowed to build a list in which the same value can appear several times. Such a collection simply is a **list**. There are many situations where lists are a useful modeling construct. Let us consider second-hand cars sold by a company, and for which advertisements have been published in newspapers. Potential buyers call for appointment to examine one of the cars. The calls are recorded by an answering machine, then, later on, the callers are contacted in the calling order. It is not uncommon that the same person call more than once. This protocol can be modeled by Figure 9.19. The car for which the phone calls are

illustrated have been (or will be) examined by potential buyer with number 75.83.12, then by the one with number 22.67.40, and finally by the first one again, exactly in this order.

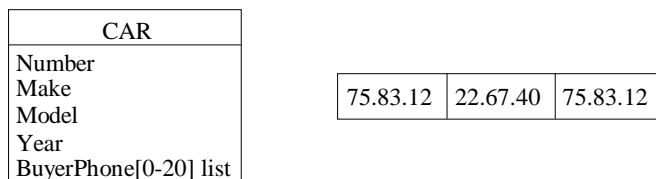


Figure 9.19 - BuyerPhone is a list multivalued attribute.

Arrays

Normally, we should have to close the discussion on multivalued attributes with the List construct. Unfortunately, we cannot. The problem is simple to state, though it is a bit more complicated to solve.

When examining what standard technologies propose to implement collections of values, we quickly learn that most of them offer one construct only, i.e., the **array** of elements. This is a universal structure, through which one can easily represent sets, bags and all sorts of lists, but which also makes it possible to implement other sorts of data structures, such as chains, hash tables, sparse tables, vectors, matrices, etc. Most decent 3GL languages such as COBOL, PL/1, ADA, BASIC, Pascal and C include some variant of the array construct⁴.

An array is not a data structure as we have discussed them so far. It is a memory organization, made of an indexed collection of cells. Each cell is designated through its position (generally an integer number starting from 1). A cell is empty until a value is explicitly stored in it. Two cells can contain the same value.

Considering the contents of these cells only, an array appears as a *list of non-unique values that can include some null-values*. The problem is that the position of a cell can be an implicit information. Figure 9.20 represents the model of a department for which we record the expenses according to 4 different

4. Some languages offers the list construct (LISP, PROLOG) while OO-DBMS often propose bag and set constructs.

categories, numbered from 1 to 4. In the sample data, there is an amount for categories 1, 2 and 4 only.

DEPARTMENT
Name
Location
Expense[0-4] array

1,250	825		1,250
-------	-----	--	-------

Figure 9.20 - Expense is an **array** multivalued attribute.

Unique arrays

Arrays being pure storing structures, there generally is no uniqueness constraints on the cell contents. Hence the concept of **unique array**. The schema of Figure 9.21 represents teams of persons in charge of projects. Each team comprises from two to four persons, each of them taking a specific role (numbered from 1 to 4) in the team.

TEAM
Code
Skill
Role[2-4] u-array

J. Barrie	Dodgson		Milne
-----------	---------	--	-------

Figure 9.21 - Role is a **unique array** multivalued attribute.

Summary

The table below shows the classification of multivalued constructs according to two dimensions: structure and uniqueness.

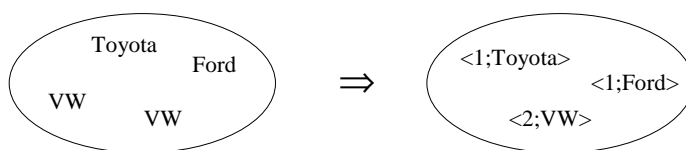
	unstructured	ordered	indexed array of cells
unique	set	u-list	u-array
non unique	bag	list	array

Generally (but there can be exceptions), sets, bags, lists and unique lists will be used in conceptual specifications while arrays will rather be used in logical and physical schemas.

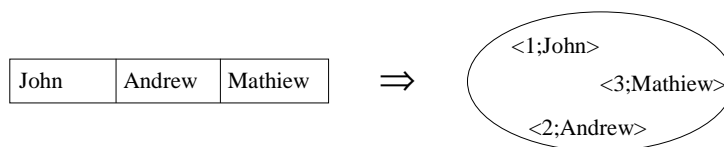
Set expression of non-set multivalued attributes

It can be shown that any non-set collection of elements can be expressed into an equivalent structure, generally more complex, but that includes set constructs only. Let us first examine how the examples of bag, list and array shown above can be transformed into set expressions.

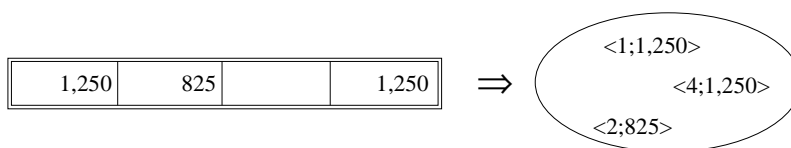
Bags. A bag is transformed into a set of pairs $\langle m ; v \rangle$, where v is a unique value and m is the number of instances of v in the bag.



Lists. A list is transformed into a set of pairs $\langle s ; v \rangle$, where s is the position of the element in the list and v its value. Note that the values of s form a continuous sequence, since each element of the list consists of a value (as opposed to arrays).

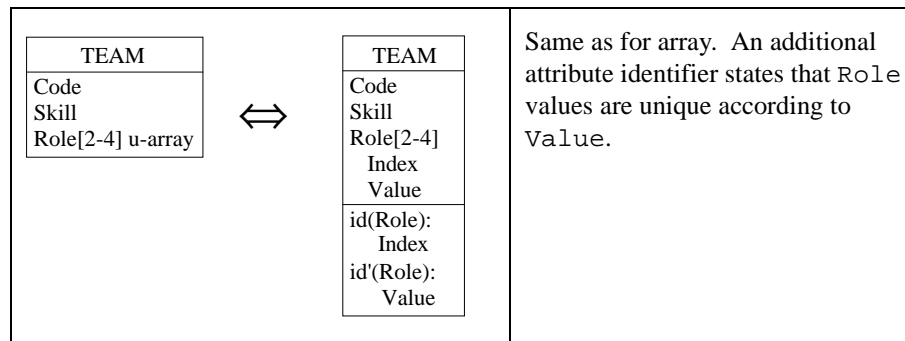


Arrays. An array is transformed into a set of pairs $\langle i ; v \rangle$, where i is the index of a non-empty cell and v is its contents. Note that the values of i do not form a continuous sequence, since empty cells are not represented.



Now, we can propose precise translation rules for all the non set constructs. For each of them, the following table shows an example of each non set attribute, its set equivalent and a short description.

<table border="1"> <tr><td>CUSTOMER</td></tr> <tr><td>CNum</td></tr> <tr><td>Name</td></tr> <tr><td>Car[0-10] bag</td></tr> </table>	CUSTOMER	CNum	Name	Car[0-10] bag	\Leftrightarrow	<table border="1"> <tr><td>CUSTOMER</td></tr> <tr><td>CNum</td></tr> <tr><td>Name</td></tr> <tr><td>Car[0-10]</td></tr> <tr><td>Multiplicity</td></tr> <tr><td>Value</td></tr> <tr><td>id(Car):</td></tr> <tr><td>Value</td></tr> </table>	CUSTOMER	CNum	Name	Car[0-10]	Multiplicity	Value	id(Car):	Value	<p>The bag of cars is replaced with a set of distinct car values. To recover the fact that each car Value can appear more than once, a Multiplicity attribute states how many times this value appears.</p>				
CUSTOMER																			
CNum																			
Name																			
Car[0-10] bag																			
CUSTOMER																			
CNum																			
Name																			
Car[0-10]																			
Multiplicity																			
Value																			
id(Car):																			
Value																			
<table border="1"> <tr><td>CUSTOMER</td></tr> <tr><td>CNum</td></tr> <tr><td>Name</td></tr> <tr><td>ChrName[1-4] u-list</td></tr> </table>	CUSTOMER	CNum	Name	ChrName[1-4] u-list	\Leftrightarrow	<table border="1"> <tr><td>CUSTOMER</td></tr> <tr><td>CNum</td></tr> <tr><td>Name</td></tr> <tr><td>ChrName[1-4]</td></tr> <tr><td>Sequence</td></tr> <tr><td>Value</td></tr> <tr><td>id(ChrName):</td></tr> <tr><td>Sequence</td></tr> <tr><td>id'(ChrName):</td></tr> <tr><td>Value</td></tr> </table>	CUSTOMER	CNum	Name	ChrName[1-4]	Sequence	Value	id(ChrName):	Sequence	id'(ChrName):	Value	<p>The list is transformed into a set of couples made up of a christian name value (Value) and a Sequence number that states the position of the value. The attribute identifiers are necessary to tell that, for each CUSTOMER entity, the ChrName values are unique according to Sequence and according to Value.</p>		
CUSTOMER																			
CNum																			
Name																			
ChrName[1-4] u-list																			
CUSTOMER																			
CNum																			
Name																			
ChrName[1-4]																			
Sequence																			
Value																			
id(ChrName):																			
Sequence																			
id'(ChrName):																			
Value																			
<table border="1"> <tr><td>CAR</td></tr> <tr><td>Number</td></tr> <tr><td>Make</td></tr> <tr><td>Model</td></tr> <tr><td>Year</td></tr> <tr><td>BPhone[0-20] list</td></tr> </table>	CAR	Number	Make	Model	Year	BPhone[0-20] list	\Leftrightarrow	<table border="1"> <tr><td>CAR</td></tr> <tr><td>Number</td></tr> <tr><td>Make</td></tr> <tr><td>Model</td></tr> <tr><td>Year</td></tr> <tr><td>BPhone[0-20]</td></tr> <tr><td>Sequence</td></tr> <tr><td>Value</td></tr> <tr><td>id(BPhone):</td></tr> <tr><td>Sequence</td></tr> </table>	CAR	Number	Make	Model	Year	BPhone[0-20]	Sequence	Value	id(BPhone):	Sequence	<p>The list is transformed into a set of pairs made up of a BuyerPhone value (Value) and a Sequence number that states the position of the value. The attribute identifier tells that, for each CAR entity, the BPhone values are unique according to Sequence.</p>
CAR																			
Number																			
Make																			
Model																			
Year																			
BPhone[0-20] list																			
CAR																			
Number																			
Make																			
Model																			
Year																			
BPhone[0-20]																			
Sequence																			
Value																			
id(BPhone):																			
Sequence																			
<table border="1"> <tr><td>DEPARTMENT</td></tr> <tr><td>Name</td></tr> <tr><td>Location</td></tr> <tr><td>Expense[0-4] array</td></tr> </table>	DEPARTMENT	Name	Location	Expense[0-4] array	\Leftrightarrow	<table border="1"> <tr><td>DEPARTMENT</td></tr> <tr><td>Name</td></tr> <tr><td>Location</td></tr> <tr><td>Expense[0-4]</td></tr> <tr><td>Index</td></tr> <tr><td>Value</td></tr> <tr><td>id(Expense):</td></tr> <tr><td>Index</td></tr> </table>	DEPARTMENT	Name	Location	Expense[0-4]	Index	Value	id(Expense):	Index	<p>Each non-empty cell of the array is expressed as an Expense value, that comprises the Index value of the cell and the Value stored in it. The attribute identifier tells that, for each DEPARTMENT entity, the Expense values are unique according to Index.</p>				
DEPARTMENT																			
Name																			
Location																			
Expense[0-4] array																			
DEPARTMENT																			
Name																			
Location																			
Expense[0-4]																			
Index																			
Value																			
id(Expense):																			
Index																			



Big question: do we have to know all these rules? Not at all, DB-MAIN know them much better than we could ever. For instance, select attribute ChrName of entity type CUSTOMER and execute command **Transform / Attribute / Multi Conversion**. We get a fairly complex control panel (Figure 9.22) which asks us into what kind of collection type we want attribute ChrName to be converted. There seems to be a lot of possible target structures, but few conversions are labelled **no loss**, stating that we will loose no information if we select them.

To understand this panel we must examine some outstanding properties of multivalued attributes. Three of them are of particular importance, namely uniqueness, order and possible gaps.

1. **Uniqueness.** Three constructs enforce uniqueness on their elements, namely sets, unique lists and unique arrays. The other three accept multiple instances of the same value.
2. **Order.** Two constructs form unordered collections of values, namely sets and bags. The other four induce an order on the collection of their elements.
3. **Gaps.** A cell of an array (be it unique or not) can be empty, leaving a gap between its adjacent cells. This gap can have a specific meaning (absent value for instance). Sets, bags and lists ignore this concepts.

When converting a collection type into another one, each of these properties can be preserved (denoted by a green =), lost (denoted by a red -) or introduced (denoted by a red +). A conversion produces a equivalent construct if it preserves all three properties (presence or absence), i.e., if it is characterized by three green = in the control panel. For a unique list, only the second conversion preserves all its properties (Figure 9.22).

The best way to understand all this is to play with the different kinds of multivalued attributes and to apply all the possible conversions.

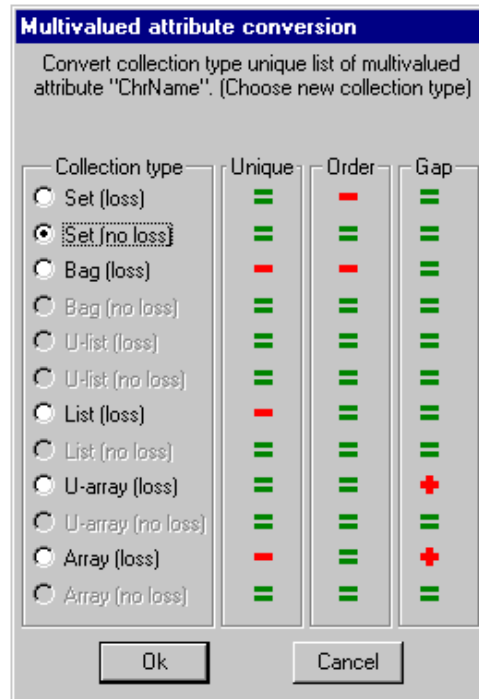


Figure 9.22 - Converting non-set multivalued attribute ChrName of CUSTOMER into another collection type (including set).

9.7 Multivalued identifiers

Identifiers comprise attributes and/or roles. However all the examples used so far were based on single-valued attributes. Nothing prevents us from defining an identifier with a multivalued attribute, such as in the schema of Figure 9.23 (left), which states that customers have from 0 to 10 account numbers. Not only each customer has unique account numbers (which is quite natural since we defined them as a set of values), but each account number is unique among all customers. In other words, an account number belongs to one and only one customer.

This property is simply declared through the secondary identifier {Account[*]}. Note that the bracket part of expression Account[*] tells that each value of Account is, in its own right, an identifier.

To better grasp the essence of this construct, let us transform attribute Account into an entity type (Figure 9.23, right):

1. we select attribute Account,
2. we call command **Transform / Attribute / -> Entity type**,
3. we validate all the propositions (except for the rel-type name which will feel better when renamed as "of").

Now look very carefully at both schemas, and try to convince yourself that they convey exactly the same semantics.

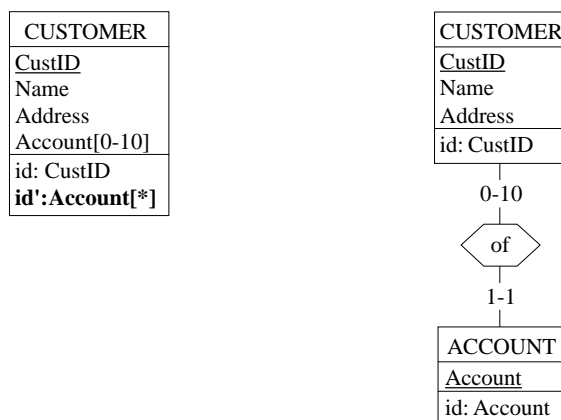


Figure 9.23 - In the left side schema, attribute Account has been declared an identifier of CUSTOMER to represent the fact that *an account belongs to one customer only*. Another equivalent way to represent the situation is proposed in the right side schema.

Now, if we accept multivalued attributes as components of identifiers, we could define very complex and quite obscure (and sometimes quite wrong too!) structures. So that we would be wise to limit the valid arrangements to meaningful combinations, or at least to those that can make sense for most of us. Therefore, we propose to define identifiers that comply with the valid forms described in Figure 9.24.

Some additional rules for identifiers of entity type E:

- The attribute components of any identifier of E are attributes of E , of any level.
- Each role component r belongs to binary rel-type R such that (1) r is played by F , (2) the other role s is played by E , (3) $R.s$ has cardinality $[I-1]$, where $I=1$ for primary identifiers, (4) $R.r$ has cardinality $[K-L]$, with $L>1$.

	single-valued attribute	multivalued attribute	role
format 1	1 or more	0	0
format 2	1 or more	0	1
format 3	0, 1 or more	0	2 or more
format 4	0	1	0

Figure 9.24 - Valid formats for entity type identifiers.

9.8 More on access keys

The identifier of Figure 9.23 (left) can be declared an access key as well (do it!), yielding a multivalued foreign key. Of course, such complex access keys cannot be translated in a straightforward way into relational indexes⁵. But all this is another story that will be told later on.

Access keys form a particular *species*, whose habits are described in Figure 9.25.

	single-valued attribute	multivalued attribute	role
format 1	1 or more	0	0
format 2	1 or more	0	1 or more
format 3	0	0	1 or more
format 4	0	1	0

Figure 9.25 - Valid formats for access keys.

5. Anyway, some (rare) DBMSs can manage multivalued indexes. ADABAS from Software AG is one of them.

Additional rules for access keys of entity type E:

- The attribute components of any identifier of E are attributes of E, of any level.
- Each role component r belongs to binary rel-type R such that (1) r is played by F, (2) the other role s is played by E.

The schema of Figure 9.26 shows some examples of non-standard access keys. They state that one can get fast access to PERSON entities (or records) (1) from any CarID value, (2) or from any ZipCode value, (3) or from AccountNumber value of any Account value, (4) or from any REGION entity (or record) via lives_in.

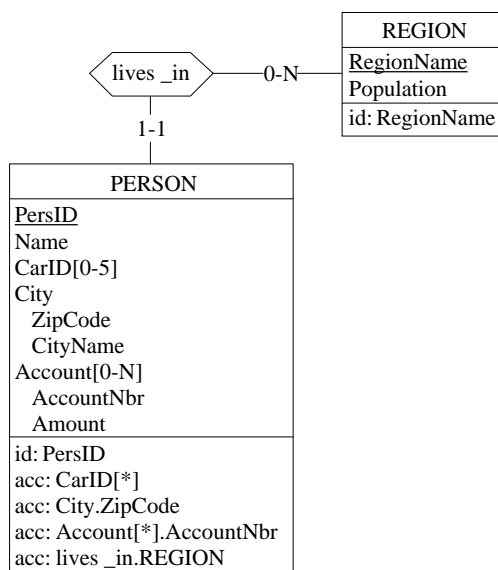


Figure 9.26 - A handful of non-standard access keys.

9.9 Multivalued reference attributes

If identifiers and access keys can be multivalued, why couldn't foreign keys be multivalued as well? They can, indeed. Figure 9.27 shows that the link between CUSTOMER and ORDER can be defined by multivalued foreign key CUSTOMER.Passes from CUSTOMER to ORDER, instead of the more traditional single-valued foreign key from ORDER to CUSTOMER.

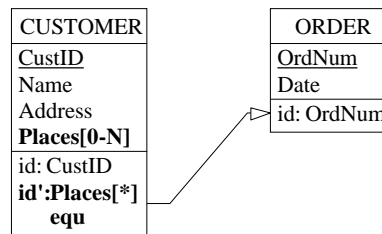


Figure 9.27 - A multivalued foreign key. Note that the **id'** constraint states that an ORDER entity cannot be referenced by more than one CUSTOMER entity and the **equ** constraint states that each ORDER entity must be referenced by one CUSTOMER entity.

The schema of Figure 9.28 is a more sophisticated illustration of the concept. The multivalued foreign key implements a *many-to-many* rel-type between UNIT and PRODUCT.

Multivalued foreign keys will be found either in standard files, where record types can include multivalued fields acting as implicit, i.e., undeclared, foreign keys, or in modern RDBMSs (SQL-3 or SQL:1999), that provide some way to define multivalued columns. However, it can be demonstrated that any data structure, even apparently purely relational, can include implicit multivalued foreign keys. Indeed, a single-valued field can result from the concatenation of the values of a multivalued field, and therefore represent potential multivalued foreign key. Such complex structures are studied in the theory of reverse engineering and are beyond the scope of this tutorial. However they are worth being mentioned.

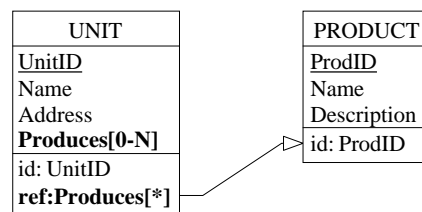


Figure 9.28 - A multivalued foreign key that implements a many-to-many link between two entity types: *a production unit can produce several products while a product can be produced by several units.*

9.10 Non-standard reference attributes

In the previous section, we already shook the apparent simplicity of the concept of foreign key by suggesting that they could be made up of a multivalued attribute. Not surprisingly, they can comprise compound or sub-attributes as well. In addition, we will mention some other curious forms of foreign keys. Some of them will appear naturally in standard database analysis and design while others will be found in legacy databases.

- **Optional FK.** All the components of the foreign key are optional; if the latter comprises more than one component, the foreign key should form a coexistence group as well (why?).
- **Total FK.** A foreign key is called total if each value of the target identifier must be referenced by at least one source entity. Such a foreign key has been described in Section 6.4, and is represented by symbol eq .
- **Cyclic FK.** A cyclic foreign key references its own table instead of another table, so that it is a bit less *foreign* than the standard FK.
- **Identifier FK.** The foreign key is an identifier as well.
- **Secondary FK.** The foreign key references a secondary identifier instead of a primary identifier.
- **Conditional FK.** The components can be interpreted as a foreign key only under a definite condition.
- **Multi-target FK.** The foreign key references more than one table. Each value designates a row in each of the target tables.
- **Alternate FK.** The foreign key references more than one table. Each value designates a row in one of the target tables only.

However, some even stranger kinds of foreign keys can be encountered in the database jungle. We will examine two of them, that must be interpreted at the logical level⁶, so that we will talk about record types (these structures are not relational) instead of entity types and about fields instead of attributes. Five more examples are proposed in the Exercise section.

6. About 25 non-standard foreign key patterns are discussed in J-L Hainaut, J-M. Hick, J. Henrard, V. Englebert, D. Roland, *The Concept of Foreign key in Reverse Engineering - A Pragmatic Interpretative Taxonomy*, DB-MAIN Research Report, March 1997, FUNDP.

Hierarchical foreign key to a multivalued attribute

Record types, as they appear in standard files, often compensate the lack of inter-record explicit relationships by complex intra-record hierarchical field structures. In particular, multivalued compound fields, possibly at several levels, are popular structures to implement a hierarchy of entity types. In such a structure, some dependant entities can be represented by instances of multivalued fields, instead of by records. Referencing these entities from within other records consists in designating these values. Hence the concept of foreign keys referencing field values instead of records.

In the schema of Figure 9.29, an ORDER record represents a customer order that includes from 0 to 20 details. Each of these details mention a different item in a definite quantity. This structure is represented by the ORDER record type which includes multivalued field `Detail`. This field has distinct `ItemCode` values (this property is declared by an attribute identifier). To identify a unique `Detail` value, the programmer must supply a values of `OrdID` and a value of `ItemCode`. For each detail, some shipments can be sent to the customer. Therefore, each shipment is associated with a detail. Each SHIPMENT record designates its parent `Detail` value through the hierarchical foreign key `{OrdID, ItemCode}`.

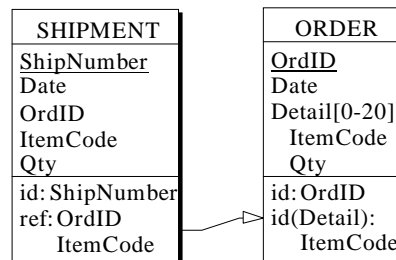


Figure 9.29 - The foreign key references a value of a multivalued attribute, instead of an entity.

Overlapping foreign keys

Two multi-component foreign keys overlap if they share one or several columns and if none is a subset of the other. The schema of Figure 9.30 describes lines of invoice, each of which belongs to an invoice and references a line of order. Both invoice and line of order reference their unique origin order, hence common component `OrderNumber`.

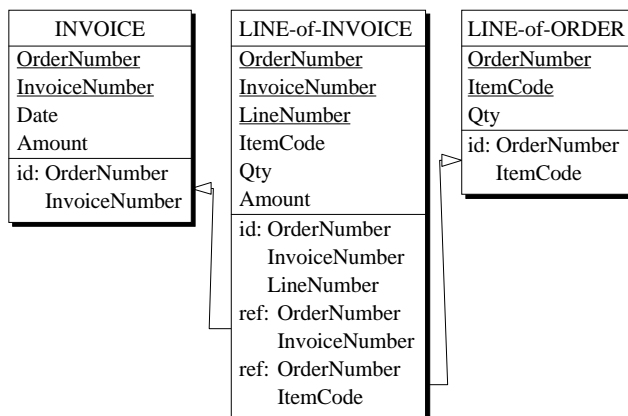


Figure 9.30 - The overlapping foreign keys share common field OrderNumber

9.11 Object attributes

As has been accepted until now, the domain of an attribute is a set of values. Some domains are made up of atomic values (numbers, character strings, dates and the like) while others comprise more complex structures such as compound domains as illustrated in Section 9.3.

Considering that a domain is just a set of *things*, why can't these *things* be entities? Figure 9.31 is an example of this idea. Attribute Sender has been given a domain that is entity type CUSTOMER instead of any set of elementary values.

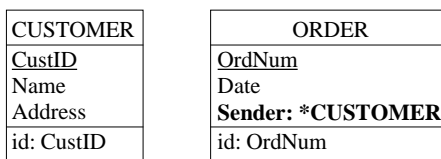


Figure 9.31 - The value of attribute Sender is not a CustID value, as we could expect, but a CUSTOMER entity!

In other words, each value of this attribute **really is an entity**, not a printable value such as CustID, as would have been the case should we declare Sender a foreign key to CUSTOMER. For instance, we can talk about sub-attributes Sender.Name and Sender.Address of ORDER, as if Sender were a compound attribute.

The schema of Figure 9.32 pushes the idea even further. The value of attribute OrderPlaced of entity type CUSTOMER is a set of ORDER entities. As above, the value of Sender is a CUSTOMER entity. Note that OrderPlaced is an identifier of ORDER, translating the fact that an order is placed by one customer only, and therefore identifies it. Each Detail value includes a value of attribute Product, which is a PRODUCT entity.

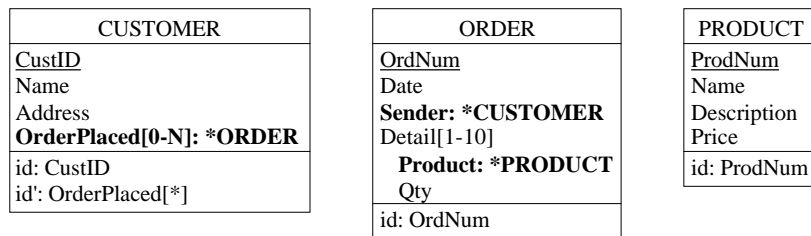


Figure 9.32 - This schema describes customers that place orders, the details of which reference products. Object attributes is an alternative to rel-types and foreign keys.

This schema includes a new feature, namely *redundant structures*. Indeed, telling who is the sender of each order gives the same information as designating all the orders of each customer. Therefore, object attributes OrderPlaced and Sender convey exactly the same information. Moreover, they can be considered as the inverse of each other. When customer C places order O, we must add entity O to attribute OrderPlaced of entity C and attribute Sender of O must be set to C. Declaring that Sender and OrderPlaced are inverse object attributes can be made as in Figure 9.34. To state this constraint, we proceed as follows.

1. Each attribute must form a group. That is already done for OrderPlaced, so that we just select Sender and we click on button GR in the Standard tools bar.
2. We open the Property box of any of both groups (say, Sender) and we click on button Constraint to call the Inter-group constraint panel.

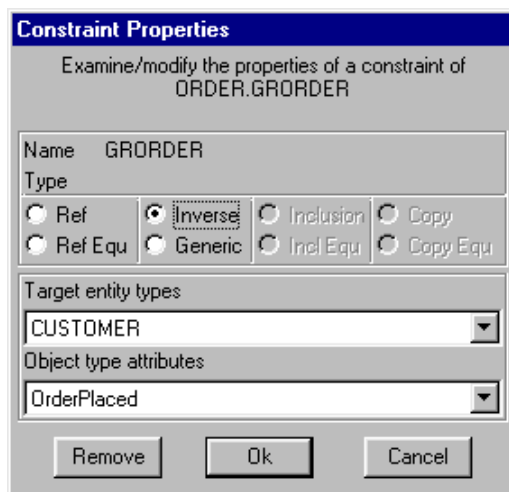


Figure 9.33 - Choosing an inverse object attribute.

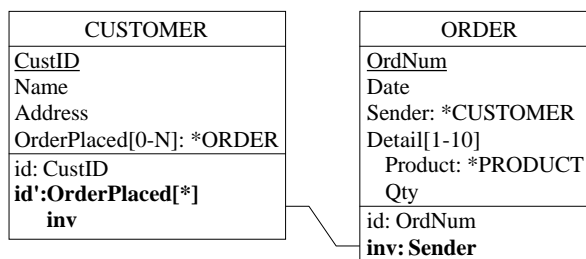


Figure 9.34 - Object attribute `OrderPlaced` has been declared the inverse of `Sender` of `ORDER` to express the fact that *the sender of an order placed by customer C is C him/herself*.

3. Button `Inverse` is already checked (DB-MAIN understands what we are doing). All the candidate inverse object attributes are shown in fields `Target entity types` and `Object type attributes`⁷. We select one of them and we validate the choice.

The schema appears as in Figure 9.34.

For what reason could we want to represent inter-entity relationships through object attributes instead of pure rel-types? Because it is the preferred (and only) way to link object classes in **object-oriented DBMSs** or OO-DBMSs (where entity types are interpreted as object classes). In **object-relational DBMSs**, or ORDBMS, we can use either foreign keys or table-based columns, i.e., object attributes.

The concept of inverse object attribute will be useful in OO-DBMSs, where object attributes are often used to navigate between object classes: to get the sender of an order **and** the orders placed by a customer. On the contrary, OR-DBMSs do not require such doubled representations. To close the discussion, we give in Figure 9.35 a pure Entity-relationship schema that is equivalent to that of Figure 9.32.

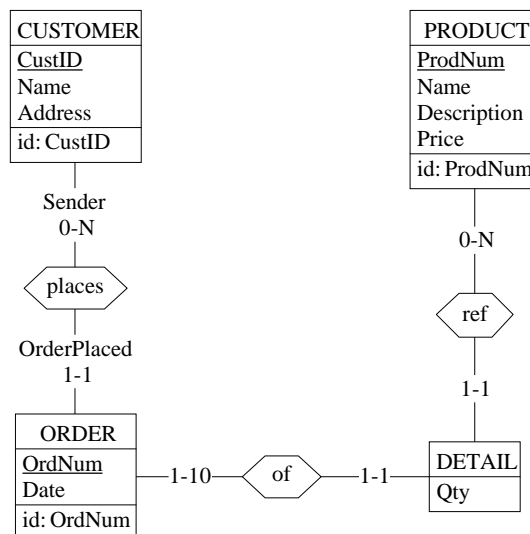


Figure 9.35 - An Entity-relationship schema equivalent to that of Figure 9.32.

-
7. If the inverse attribute you have in mind is not displayed, that means that it cannot be declared the inverse of the current attribute. For instance, `OrderPlaced` appears in the candidate list because its domain is `ORDER` and it has been declared an identifier of `CUSTOMER`. If one of these conditions is not met, `DB-MAIN` does not propose this attribute.

Key ideas of Lesson 9

1. Built-in and user-defined domains

Built-in domains are the basic data types which atomic attributes can be based on in DB-MAIN. They correspond to the data types available in most DBMSs and programming languages. Additional, application-specific, domains can be defined as a named combination of basic data types. They offer a nice way to build a collection of reusable components on which the schemas can be built.

2. Stable and non-recyclable attributes

The value of a stable attribute can be set, but cannot be changed afterwards. If an attribute is non-recyclable, then any value that was once given to the attribute of an entity cannot be assigned to another entity, even when the former has disappeared.

3. Attribute identifiers

Considering multivalued, compound, attribute A depending on parent P (entity type, rel-type or compound attribute), a subset of the components of A can be declared an identifier for A. For each parent instance, the values of A are unique on this subset of values.

4. Non-set multivalued attributes

In many situations, the value of a multivalued attribute is a *set* of values, that is, an unordered collection of distinct values. Sometimes, we need more sophisticated kinds of value collections. A *bag* is an unordered collection of values that are not necessary distinct. A *unique list* is an ordered collection of distinct values, while these values can be nonunique in a simple *list*. An *array* is an indexed set of cells in which values can be stored. In a *unique array*, these values are unique. Note that some cells can be left empty.

Each non-set collection type can be transformed into an equivalent standard, set-oriented, multivalued attribute.

5. Multivalued identifiers

An entity type identifier can comprise attributes and/or roles. They can also be made up of a multivalued attribute

6. Access keys

Complex access keys, defined on non-relational schemas can include multivalued attributes, component attributes and even remote roles. Such access keys will be used in optimized logical schema design.

7. Non-standard reference attributes

Besides classical relational foreign keys, made up of one or several atomic, single-valued column, and referencing the primary id of one table, many other kinds of referential structures can be encountered in actual databases. Interpreting these structures is a problem that pertain to the reverse engineering domain.

8. Object attributes

An object attribute is an attribute whose domain is an entity type. Two object attributes can be declared inverse of each other. An object attribute can be used to represent a relationship type.

Such structures will be found in OO databases, in which entity types are called object classes instead. They can be used in plain entity-relationship schemas to define domains that are more complex than user-defined domains.

Summary of Lesson 9

- In this first lesson, we have studied the following concepts:
 - Built-in domains
 - User-defined domains
 - Stable and non-recyclable attributes
 - Attribute identifiers
 - Non-set multivalued attributes
 - Multivalued identifiers
 - Complex access keys
 - Multivalued reference attributes
 - Non-standard reference attributes
 - Object attributes

- We have also learned
 - to define and to use user-defined domains
Product / User domains
 - to define an attribute identifier
select the attribute, then **New / Group**
 - to transform an attribute into an entity type
Transform / Attribute / -> Entity type
 - to transform an entity type into an attribute
Transform / Entity type / -> Attribute
 - to convert non-set multivalued attributes into set attributes
Transform / Attribute / Multi Conversion

Exercises for Lesson 9

- 9.1 Define a dozen attribute that are specific to an accounting system. Same exercise for an application domain that talks about students, teachers, lectures and exams.
- 9.2 Transform the following structure into a schema that does not include multivalued attributes.

CUSTOMER
<u>CustID</u> Name Purchase[0-100] Date Item Qty Shipment[0-5] Date Sty
id: CustID id(Purchase): Item Date id(Purchase.Shipment): Date

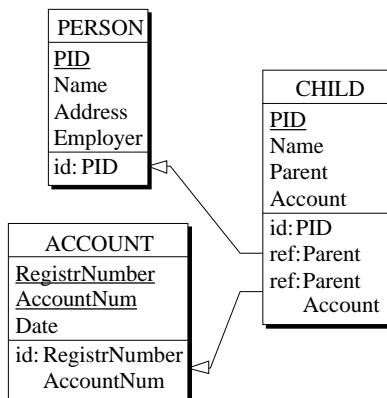
- 9.3 Transform each example of non-set attribute proposed in this lesson into a pure relational schema.

Hint: first transform it into a set multivalued attribute, then transform the latter into relational structures through command **Transform / Relational model**.

Compare the result with the source schema.

- 9.4 **Embedded foreign key**

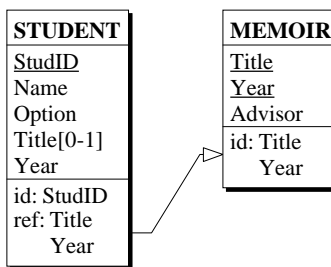
In a social security system, each child depends on a parent (which is a person), and is associated with an account. A developer has coded this situation as follows.



What do you think of this schema? Can you restructure it to make its actual semantics explicit?

9.5 Partly optional foreign key

A collection of memoir subjects are proposed to last year students. A memoir is identified by its title and the year it is being proposed. Students are characterized by their name, the option they are registered in, and the year they have to choose a memoir subject. When they have made this choice, they are given the title of their memoir. Technically speaking, when attribute `Title` of a `STUDENT` entity is null, then this entity references no `MEMOIR` entity, while when `Title` is not null, then $\{\text{Title}, \text{Year}\}$ references a `MEMOIR` entity.

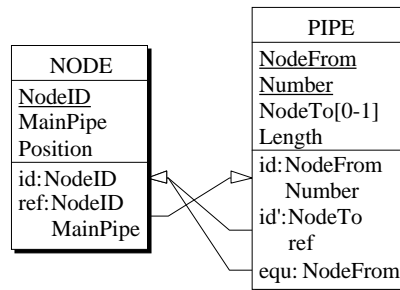


This structure cannot be declared a foreign key in every RDBMS. Propose another equivalent structure that can be fully coded in SQL.

By the way, could you give a correct Entity-relationship schema of this implementation?

9.6 **Partially reciprocal foreign keys**

A fluid distribution network is made up of nodes and pipes linking nodes in a directed tree structure. The fluid flows from the root node to the leaf nodes. In a pipe, it flows from the source node to the sink node. The pipes attached to a common source node are uniquely numbered. Among the outgoing pipes of each source node, one is considered its main pipe. A developer proposes the following relational schema, that comprises a non-standard foreign key pattern, called *partially reciprocal FK*. What could be the conceptual schema it is an implementation of?



9.7 Design a relational schema that describes the following application domain: *Towns are situated in countries (or states). Towns in the same country have distinct names. In each country, one town is known as its capital.*

Give an equivalent conceptual schema of this logical schema.

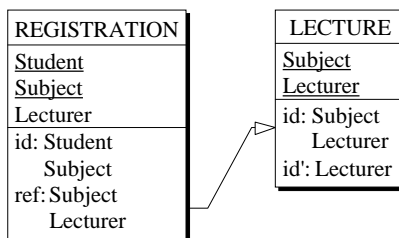
9.8 **Non-minimal FK**

Many text books about relational theory base their introduction to Boyce-Codd normal form on the following example:

```

registration(Student, Subject, Lecturer)
Lecturer --> Subject
Student, Subject --> Lecturer
    
```

There are several ways to transform this schema. One of them could be as follows:

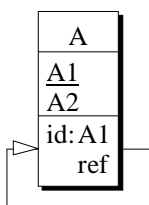


A LECTURE entity represents the fact that a lecturer teaches a given subject. A lecturer is allowed to teach one subject only. This fact is notified by the secondary identifier {Lecturer}. A (non-minimal) primary identifier comprising {Subject, Lecturer} has been defined for technical reasons that have to be discovered. A REGISTRATION entity states that a student is entitled to be taught a subject by a lecturer. The identifier of REGISTRATION enforces the following constraint: a student can be taught a subject by one lecturer only.

Can you explain the rationale of unusual identifier {Subject, Lecturer} and foreign key {Subject, Lecturer}?

9.9 Reflexive foreign key⁸

What do you think of the following schema? Can it be simplified?



8. This pattern was really found in an Oracle database (probably generated by a CASE tool). We are still trying to guess what it was intended for!

Lesson 10

More about constraints

Objective

This lesson describes new constraints to better express the complexity of application domains, namely existence constraints. These constraints dictate, among others, which attributes of an entity type must have a value while others cannot. We will show that these constructs are strongly related to IS-A relations. More general forms of constraints can be declared, namely the generic constraints.

In addition, this lesson will introduce to the powerful concept of schema transformation.

10.1 Introduction

This lesson will discuss new forms of constraints that can be used to better capture the semantic structures of complex application domains. We start DB-MAIN and we create a new project called Lesson10.

10.2 Existence constraints

When you observe, in the application domain, that

- *an employee can be the manager of a department or the head of a project but not both,*
- *when an employee works in a department, then the date s/he was hired is known,*
- *a car must belong to a customer or allocated to a service*

you can translate these observations into **existence constraints**.

These constraints are properties that hold among groups of optional attributes and/or roles related to an entity type. They tell which of these attributes (and roles) must have a value and which ones must have, or can have, no values. We will describe in detail four of them: *coexistence*, *exclusive*, *at-least-one* and *exactly-one*.

10.3 Coexistent components of an entity type

We create a new schema, called `Coexistence`, in which we will describe persons who may work in companies and who may be married (a fairly common combination). More precisely, each person is described by its personal number, its name, the name of his/her spouse, the date s/he was married, the company s/he works for, and the date s/he was hired by this company.

However, not all the persons are married and/or work in a company. Therefore, attributes `SpouseName`, `DateMarried` and `DateHired` are optional and role `works-in.PERSON` is optional too. The corresponding schema looks like Figure 10.1.

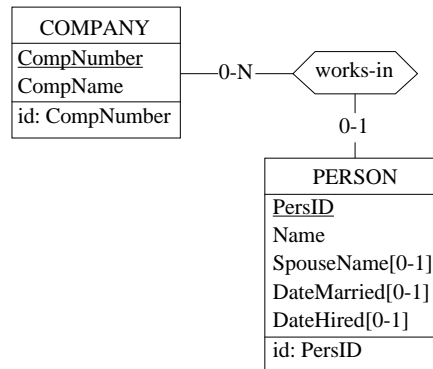


Figure 10.1 - A schema describing persons working in companies.

However, things are not so simple. For instance, all *married persons* have both valid *date married* and valid *spouse name* properties, while *non-married persons* have neither of them.

Similarly, *working persons* have a *date hired* property and a *company* they work in, while *non-working persons* have neither.

We can say that attributes `DateMarried` and `SpouseName` are **coexistent**, i.e., some entities have a value for these attributes, while all the others have no values for them.

DB-MAIN provides us with a specific feature to declare this coexistence constraint: the **coexistence** group. It works as follows:

- we create a group¹ comprising attributes `SpouseName` and `DateMarried`, and we give it the *coexistence* characteristics by clicking on the `Coexistence` button in the `Group` box (Figure 10.2);
- similarly, we define `works-in.COMPANY` and `DateHired` as another coexistence group.

The completed schema is shown in Figure 10.3 and in Figure 10.4.

1. Proceed as usual: select all the components then click on button `GR` in the Standard tools bar. To open a selected group, just press the `Enter` key.

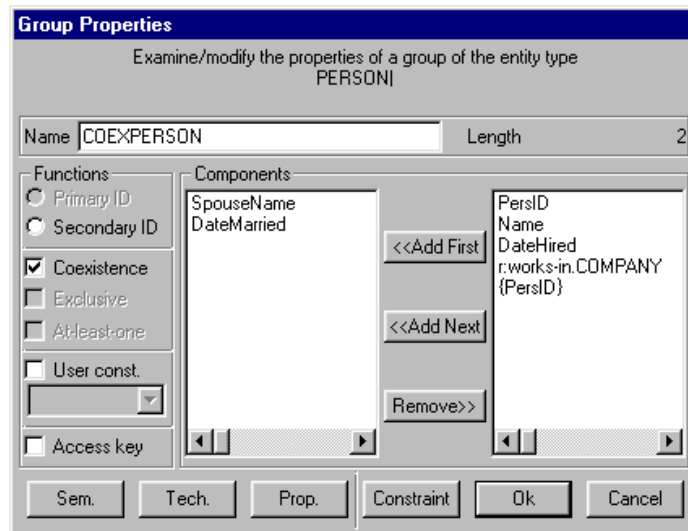


Figure 10.2 - Defining a coexistence group.

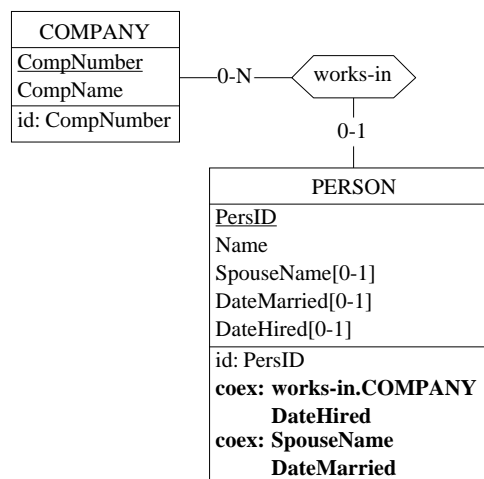


Figure 10.3 - Any person who works in a company must have a date hired, and conversely. All married persons, and only they, have a spouse name and a date of marriage.

```

COMPANY
  CompNumber
  CompName
  id: CompNumber
PERSON
  PersID
  Name
  SpouseName[0-1]
  DateMarried[0-1]
  DateHired[0-1]
  id: PersID
coexist: works-in.COMPANY, DateHired
coexist: SpouseName, DateMarried

works-in (
  [0-N]: COMPANY
  [0-1]: PERSON )

```

Figure 10.4 - Text view of coexistence constraints.

Note

1. All the components of a coexistence group must be optional. This condition is easy to check for attributes: their cardinality must be [0-j]. For the role components (e.g., `works-in.COMPANY`), the rule is a bit different: the role specifies a relationship type whose other role must be optional, i.e. it has cardinality [0-1]. This rule can be explained by the following interpretation: *a PERSON optionally (i.e., [0-1]) works-in a COMPANY.*
2. A coexistence group can also be defined among the attributes of a relationship type.

10.4 Exclusive components of an entity type

This concept is quite similar to the coexistence of components.

Let us record in the current schema information about the *wages* of the persons. Considering that some persons are paid on an hourly basis, while the others are paid at the end of each month, we can define two attributes, namely `HourlyWages` and `MonthlyWages`.

However, no PERSON entity can have a value for both attributes. We consider these attributes as *exclusive*.

It is fairly easy to define an **exclusive constraint** in DB-MAIN through an *exclusive group*:

1. we create a new group² comprising attributes HourlyWages and MonthlyWages,
2. we give it the exclusive characteristic by clicking on the Exclusive button in the Group box.

The schema appears as in Figure 10.5.

Let us now consider an additional rule, stating that *companies do not hire married persons*³. In other words, a person is married, or works in a company, (or none), but not both.

PERSON
<u>PersID</u>
Name
SpouseName[0-1]
DateMarried[0-1]
DateHired[0-1]
MonthlyWages[0-1]
HourlyWages[0-1]
id: PersID
coex: works-in.COMPANY DateHired
coex: SpouseName DateMarried
excl: MonthlyWages HourlyWages

Figure 10.5 - A person paid monthly cannot be paid per hour, and conversely.

The information concerning the marriage is gathered into a coexistence group {SpouseName, DateMarried} while the information related to the pro-

-
2. Provided no such group already exists. In such a case, just double-click on it and proceed as told in step 2.
 3. *Non-equal-opportunity* companies must be modeled as well. Whether describing politically incorrect situations is politically correct or not is beyond the scope of this introduction.

fessional activity of the person is represented by the coexistence group {works-in.COMPANY, DateHired}.

The **exclusive constraint** is defined by an *exclusive group* as follows:

1. we declare a new group comprising group {works-in.COMPANY, DateHired} and group {SpouseName, DateMarried}⁴,
2. we give it the *exclusive* characteristic by clicking on the Exclusive button in the Group box.

We get the schema of Figure 10.6.

Notes

1. All the components of an exclusive group must be optional.
2. An exclusive group can also be defined among the attributes of a relationship type.
3. A simpler expression will be proposed in the following (Figure 10.10).

PERSON
<u>PersID</u>
Name
SpouseName[0-1]
DateMarried[0-1]
DateHired[0-1]
MonthlyWages[0-1]
HourlyWages[0-1]
id: PersID
coex: works-in.COMPANY DateHired
coex: SpouseName DateMarried
excl: MonthlyWages HourlyWages
excl: {works-in.COMPANY DateHired} {SpouseName DateMarried}

Figure 10.6 - Married persons cannot work in a company, and conversely. A simplified expression will be discussed in the following.

-
4. Same procedure as for attributes: select the groups then click on button GR in the Standard tools bar.

10.5 Groups with *at least one*, or *exactly one*, existing component

Let us consider again the last schema. For the purpose of the demonstration, we delete exclusive group {MonthlyWages, HourlyWages}.

Now we consider that all the persons are paid, in a way or in another. In our schema, this rule translates as follows: *at least one* of the attributes HourlyWages and MonthlyWages must have a value.

This property is called the **at-least-one constraint**, and can be specified through an *at-least-one group* as follows:

1. we declare a new group {Monthly-Wages, Hourly-Wages},
2. we click on button At-least-one in the Group box.

Without surprise, we get the schema of Figure 10.7.

PERSON
PersID
Name
SpouseName[0-1]
DateMarried[0-1]
DateHired[0-1]
MonthlyWages[0-1]
HourlyWages[0-1]
id: PersID
at-1st-1: MonthlyWages
HourlyWages

Figure 10.7 - Every person must be paid, in whatever way(s)!

Very often, such a group will also be given the *exclusive* property, to declare that *one and only one component* must have a value. To state this, we open the group again and we click on the Exclusive button, so that both Exclusive and At-least-1 buttons are checked.

This condition is defined by the **Exactly-one** property (symbolized with **exact-1** in the schema) as shown in Figure 10.8.

PERSON
<u>PersID</u>
Name
SpouseName[0-1]
DateMarried[0-1]
DateHired[0-1]
MonthlyWages[0-1]
HourlyWages[0-1]
id: PersID
exact-1: MonthlyWages HourlyWages

Figure 10.8 - Every person must be paid, but in one way only.

Notes

1. All the components of an *At-least-one* group must be optional.
2. A group cannot have both *Coexistence* and *At-least-one* properties.
3. An *At-least-one* group can also be defined among the attributes of a relationship type.

10.6 Existence constraints rules

There are some logical rules that are useful to know when one defines several existence rules. Most of them are quite intuitive, but it could be useful to shed some light on them.

Let us first consider two examples.

1. It is sometimes possible to simplify a set of coexistence groups. In Figure 10.9/left, two coexistence constraints hold among the attributes of entity type PERSON. **One attribute appears in both constraints**, which makes it valid to merge the groups.

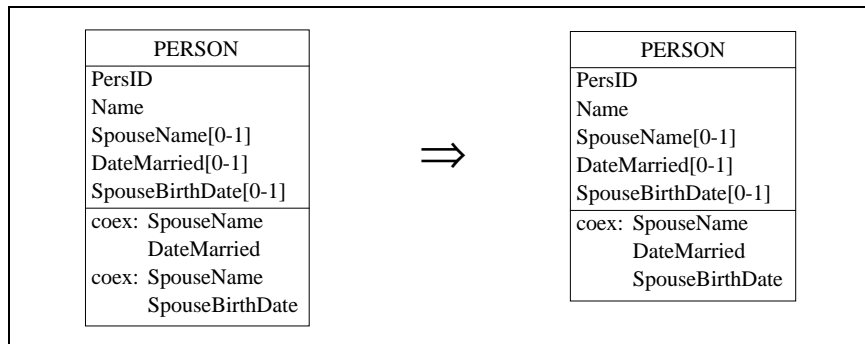


Figure 10.9 - Two coexistence constraints that share common components must be merged.

- Figure 10.6 has shown that an *exclusive* constraint can be defined on other groups. If these groups define a coexistence constraint, then the exclusive group can be reduced (Figure 10.10).

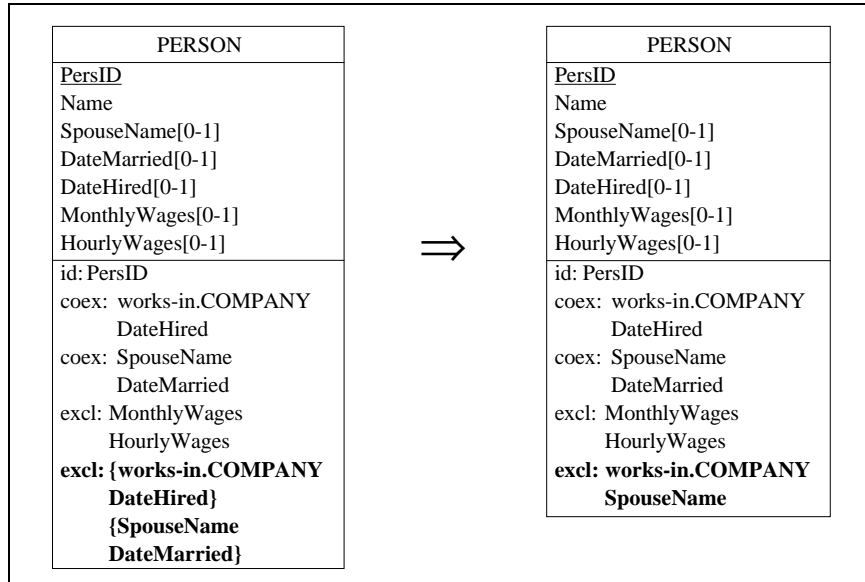


Figure 10.10 - An exclusive constraint between two coexistent groups can be simplified by replacing each group with one of its components.

It would be boring to precisely describe and illustrate all the rules that hold among any arbitrary set of existence constraints. Figure 10.11 gives some of them in an abstract way. You should easily find practical examples of each of them without too much toil. Note that some rules define inference, while others express equivalence or inconsistencies:

- $C1 \Rightarrow C2$ *Inference*: tells that whenever constraints C1 are satisfied, then constraints C2 are automatically satisfied; therefore, expressing C2 is useless.
- $C1 \Leftrightarrow C2$ *Equivalence*: tells that constraints C1 are equivalent to constraints C2: whenever one set is satisfied, the other one is satisfied as well; therefore, we can define C1 or C2.
- $C1 \Rightarrow C2 \text{ is false}$ *Inconsistencies*: tells that if constraints C1 are satisfied, then constraints C2 cannot be satisfied; therefore, expressing C1 and C2 leads to an inconsistent schema.

E	
ID	
A[0-1]	
B[0-1]	
C[0-1]	
D[0-1]	
E[0-1]	
F[0-1]	
G[0-1]	
inference rules	
coex: A, B, C	\Rightarrow coex: A, B
excl: A, B, C	\Rightarrow excl: A, B
excl: A, B excl: A, C	\Rightarrow coex: B, C
excl: A, B coex: B, C	\Rightarrow excl: A, C
coex: A, B at-least-1: A, B, C	\Rightarrow at-least-1: B, C
excl: A, B, C at-least-1: A, B	\Rightarrow excl: A, B

equivalence rules		
excl: A,B excl: A,C	\Leftrightarrow	excl: A,B coex: B,C
coex: A,B coex: A,C	\Leftrightarrow	coex: A,B,C
coex: A,B,... coex: E,F,... excl: {A,B,...}, {E,F,...}	\Leftrightarrow	coex: A,B,... coex: E,F,... excl: A,E
excl: A,B,C at-least-1: A,B	\Leftrightarrow	exact-1: A,B C <i>is always null, can be removed</i>
inconsistency rules		
excl: A,B excl: A,C	\Rightarrow	excl: B,C <i>is false</i>
excl: A,B,C	\Rightarrow	coex: B,C <i>is false</i>
coex: A,B,C	\Rightarrow	excl: B,C <i>is false</i>
at-least-1: A,B,C	\Rightarrow	at-least-1: B,C <i>is false</i>
at-least-1: A,B,C	\Rightarrow	coex: A,B,C <i>is false</i>

Figure 10.11 - Some rules to simplify and to find inconsistencies among existence constraints. This table is kindly intended for those who have problems falling asleep at night.

10.7 Existence constraints and IS-A relations

You probably found this section about *existence constraints* a bit complicated. You may even have asked yourself (from now on, you should ask us!) why *precisely these constraints* and not all the others that we can imagine. Quite right, other constraints of this kind can be defined, but these are particularly meaningful when related with the different kinds of IS-A relations. It is a bit too early to develop this point in detail, but we can get an idea on why these constraints have been privileged.

Let us consider the small IS-A hierarchy of Figure 10.12.

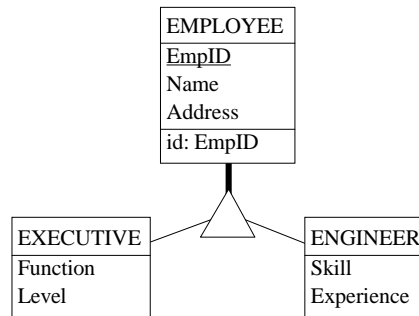


Figure 10.12 - A source schema including an IS-A hierarchy.

Now let us imagine that we want to translate this schema into relational structures. Obviously, the rules we used in Section 6.6 are useless, since they cope with simple conceptual schemas only. Though we will discuss advanced translation rules in another volume, we already can build an ad hoc relational implementation of this schema as follows (Figure 10.13).

1. We move the attributes of EXECUTIVE to EMPLOYEE. They become optional, since not all employees are executives.
2. We do the same for the attributes of ENGINEER.

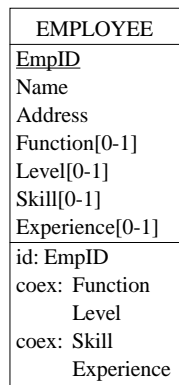


Figure 10.13 - A relational logical schema implementing the conceptual schema of Figure 10.12.

3. When an EMPLOYEE is an EXECUTIVE, s/he has values for Function and Level. Otherwise s/he has no values for them. Same reasoning for ENGINEER. Hence the coexistence constraints shown in Figure 10.13.

In Figure 10.12, no constraints held among the subtypes. No, we consider that *no executive can be an engineer*, a property that can be expressed through the D (disjoint) property of the subtypes. The implementation of Figure 10.13 can be kept, provided the additional property is explicitly expressed. It is not too complicated: if attributes {Function, Level} have a value, then attributes {Skill, Experience} must be *null*, and conversely. In short, these groups of attributes must be declared *exclusive* (Figure 10.14).

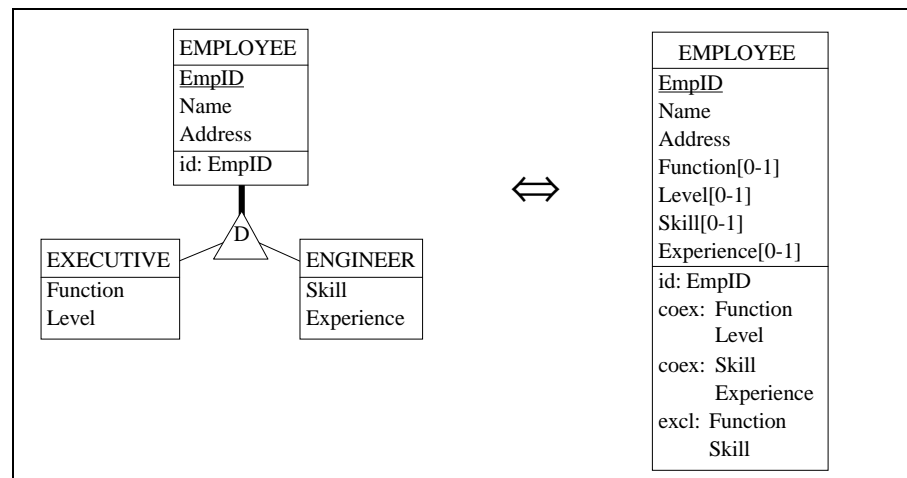


Figure 10.14 - Relational implementation of a disjunction

It is getting clear now that all the subtype properties can be completely translated into existence constraints:

- Disjoint** \Leftrightarrow exclusive
Total \Leftrightarrow at-least-one
Partition \Leftrightarrow exactly-one.

In conclusion, though existence constraints are useful in themselves, they also are necessary and sufficient to express all the subtype properties of the IS-A constructs. This stresses their importance in building correct relational logical schemas that fully translate IS-A relations, as we will see in the next volume.

10.8 Other existence constraints

The constraints described so far are the most important among existence constraints. However, many more can be imagined that cannot be explicitly included neither in formal models (such as the Entity-relationship model) or in CASE tools. The implication constraint is one of them.

In its simplest form, the implication constraint states that the value of some attributes B_1, B_2, \dots can exist only when the values of other attributes A_1, A_2, \dots exist as well. In other words, the existence of attributes B_1, B_2, \dots *implies* the existence of attributes A_1, A_2, \dots

Hence the implication expression:

$$\{B_1, B_2, \dots\} \Rightarrow \{A_1, A_2, \dots\}$$

Let us consider the schema of Figure 10.15, in which the following constraints hold:

- the *name of the spouse* of an employee is valid only when this employee is married, i.e., when s/he has a DateMarried;
- the *birthdate of the spouse* of the employee is valid only when this employee is married, i.e., when s/he has a DateMarried;

PERSON
PersID
Name
SpouseName[0-1]
DateMarried[0-1]
SpouseBirthDate[0-1]

Figure 10.15 - An entity type in which two implication constraints hold.

These constraints can be expressed as follows:

- $SpouseName \Rightarrow DateMarried$
- $SpouseBirthDate \Rightarrow DateMarried$

Unfortunately, there is no specific construct to declare such constraints, so that we are forced to imagine an equivalent structure. What about the schema of Figure 10.16? Can you prove that this schema translate both implication constraints correctly?

PERSON
PersID
Name
Marriage[0-1]
DateMarried
SpouseName[0-1]
SpouseBirthDate[0-1]

Figure 10.16 - This schema is intended to express implication constraints: *no SpouseName without DateMarried* and *no SpouseBirthDate without DateMarried*. Is it correct?

10.9 Generic constraints

The lessons studied so far have described a fairly large variety of constraints: identifiers, attribute domains, attribute cardinality, rel-type cardinality, existence, subtype properties, etc. However, it is impossible to enumerate all the constraints that are, or that may be, useful to describe some application domains. Let us consider, for example, a constraint such as the following: *no employee can receive a salary greater than that of his/her manager* or *orders can have a non-zero rebate only if they have at least 3 details*. Hence the need for a more general means to declare arbitrary kinds of constraints. DB-MAIN offers two generic constructs that can be used to define new constraints, namely the *generic group* constraint and the *generic inter-group* constraint.

Generic group constraints

Let us consider a specific constraint that tells that *among the mentioned numeric attributes, at least one must be positive*. It is a kind of *at-least-one* constraint, but this one checks the actual values, not only their presence or absence.

We will illustrate this constraint on the schema of Figure 10.18, where at least one of the three account levels must be greater than zero. The constraint will be named `at-1st-1>0`, to be interpreted as *at-least-one-greater-than-zero*.

Practically, we proceed as follows:

1. We define a group comprising the three attributes: we select these attributes and we click on the button GR.
2. We open the property box of this group: we press the Enter key.

3. We click on the button User const., we type `at-1st-1>0` in the constraint field (Figure 10.17). We close the Property box by clicking on OK.

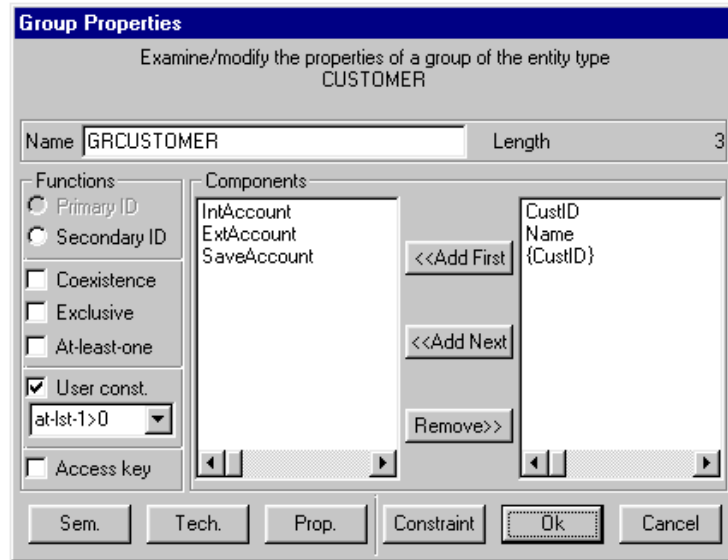


Figure 10.17 - A new form of constraint is being defined.

The constraint appears as in Figure 10.18. Once this constraint has been defined, it can be used anywhere in any project without being redefined⁵.

CUSTOMER
<u>CustID</u> Name IntAccount ExtAccount SaveAccount
id: CustID at-1st-1>0: IntAccount ExtAccount SaveAccount

Figure 10.18 - A new form of constraint tells that at least one of the attributes must have a positive value.

5. In fact, its definition has been stored in the `DB-MAIN.ini` file.

Generic inter-group constraints

Several constraints are defined among groups or from a group to another one. Constraints Ref, Equ and Inverse are some of them. The DB-MAIN model allows us to define our own inter-group constraints. Figure 10.19 shows a schema that describes *suppliers that supply items*, and *orders whose products are assigned to suppliers*. One obvious constraint is that *one cannot assign a product of an order to a supplier if this supplier does not supply this product*. In other words, all the couples {SUPPLIER,ITEM} of relationships assigned must be a subset of supplies relationship set. Such a constraint belongs to the general family of *inclusion* constraints, of which the referential constraint is just a special case.

An inter-group constraint is built as follows.

1. We define group {SUPPLIER,ITEM} of supplies and group {SUPPLIER,ITEM} of assigned;
2. The latter is opened and defined as a generic group constraint with the name incl.
3. We call the constraint panel (by clicking on button Constraint), we select the target object RT:supplies and the target group {SUPPLIER,ITEM}. We click on the button Generic and close all the panels.

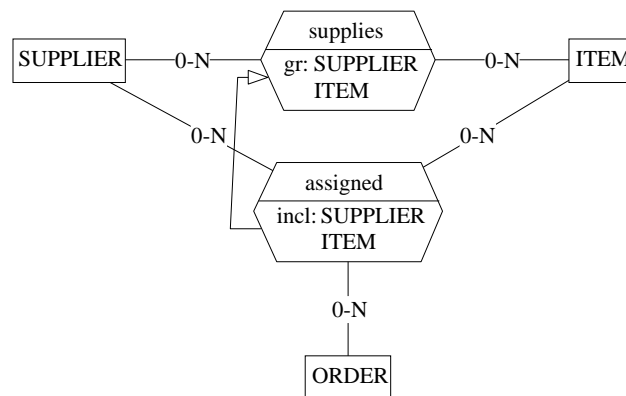


Figure 10.19 - A new inter-group constraint that tells that each couple of {SUPPLIER,ITEM} that appears in an assigned relationship must be a supplies relationship as well.

Note

Generic constraints can be considered as *passive* constraints, since the tool does not understand their semantics. They can be processed through ad hoc *Voyager* procedures.

10.10 Schema transformation: another look

To help understand the concept of *coexistence* constraint, we will propose an equivalent structure which may be more illustrative of the very nature of this constraint. To do so, we will use again the **transformation** toolkit of DB-MAIN. This component will be studied in greater detail in future lessons, but the current situation is a good opportunity to experiment with one of its simplest tools: *attribute aggregation*.

We consider the schema of Figure 10.3, and we proceed as follows:

- we select, by clicking on it, the group that comprises SpouseName and DateMarried;
- we execute command **Transform / Group / Aggregation** (Figure 10.20);
- a new attribute is created; we give it the name Marriage (or any other name);

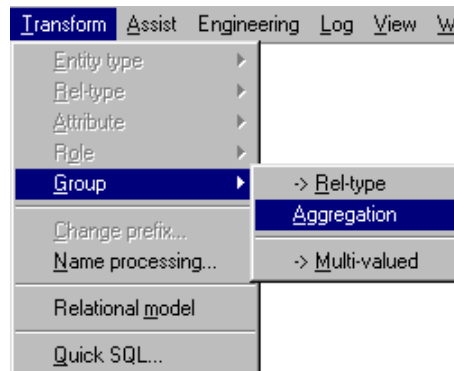


Figure 10.20 - Asking for the aggregation of the components of the selected group into a compound attribute.

As illustrated in Figure 10.21, the set of attributes of PERSON has been restructured as follows:

- now, SpouseName and DateMarried are the components of the new compound attribute Marriage;
- these attributes are mandatory for their parent attribute;
- Marriage is optional;
- the coexistence constraint has been removed.

It is important to be convinced that the schemas of Figure 10.3 and Figure 10.21 convey exactly the same semantics, i.e., they describe the same portion of the application domain. Indeed, Figure 10.21 tells that a PERSON entity can have a Marriage value. In this case, it has a value for each of its components, namely SpouseName and DateMarried. If it has no Marriage value, then, quite naturally, it has no values for the components of this attribute. This is exactly what the coexistence constraint is intended to express.

PERSON
<u>PersID</u>
Name
Marriage[0-1]
SpouseName
DateMarried
DateHired[0-1]
id: PersID

Figure 10.21 - Coexistent group {SpouseName,DateMarriage} has been transformed into optional compound attribute Marriage.

To push the experiment a bit further, we select the attribute Marriage, and we execute the command **Transform / Attribute / Disaggregate**.

(Not really) surprisingly, we get the origin schema! We can draw from this two essential conclusions that will be discussed later on:

1. each transformation is the inverse of the other one: each one erases the effect of the other one; they are called *inverse transformations*;
2. both schemas are equivalent, i.e., they represent exactly the same reality, though through different structures. The choice of one of them will be guided by criteria which are beyond the scope of this lesson. A transformation which replaces a schema with an equivalent one is called *reversi-*

ble, or semantics-preserving.

As we will see later on, such a transformation can be summarized as in Figure 10.22.

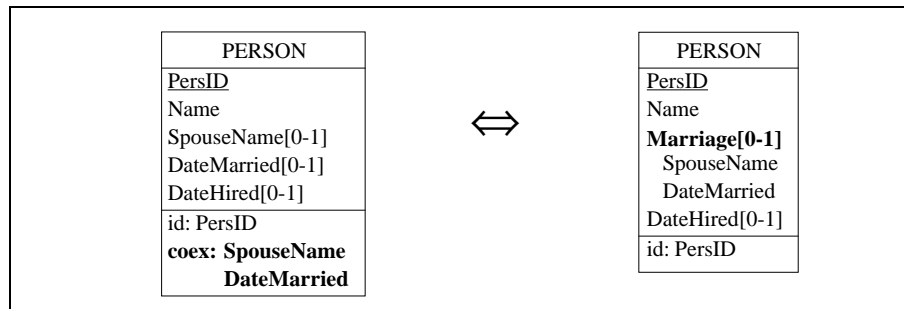


Figure 10.22 - A couple of reversible transformations: Group/Aggregate (left to right) and Attribute/Disaggregate (right to left).

DB-MAIN offers a fairly large number of schema restructuring techniques, or schema transformations. These are among the most simple, but not the least useful, as will be illustrated in further lessons.

Note

The other *coexistence* group can be processed in a similar way. However, it would need a more sophisticated transformation since it includes attributes **and roles**. Thus, we will leave it to a further lesson.

Key ideas of Lesson 10

1. Existence constraints

These constraints are properties that hold among groups of optional attributes and/or roles related to an entity type. They tell which of these attributes (and roles) must have a value and which ones must have, or can have, no values.

We considered four of them:

- *coexistence*: the components of the group must be simultaneously present or absent for any entity; the group appears with the symbol **coex**;
- *exclusive*: among the components of the group at most one must be present for any entity; the group appears with the symbol **excl**;
- *at-least-1*: among the components of the group, at least one must be present for any entity; the group appears with the symbol **at-1st-1**; all its components are optional;
- *exactly-1*: among the components of the group, one and only one must be present for any entity (= exclusive + at-least-1); symbol *exact-1*.

Existence constraints can also hold among the attributes of a rel-type. Existence constraints can translate in relational logical schemas the subtype properties (D, T) of IS-A relations.

2. Generic constraints

A generic group constraint is a user-defined property holding among the components of a group. It defines a new integrity constraint which is given a user-defined name. A generic inter-group constraint is a user-defined directed link drawn from one group to another group. The meaning of generic constraints is user-defined.

3. Schema transformation

A schema transformation is an operator that replaces constructs in a schema with other constructs. Each transformation has an *inverse* that can undo its effect. A transformation that changes the form of the schema without affecting its semantic contents is called *semantics-preserving*. Replacing a coexistent group of attributes with a compound attribute (*attribute group aggregation*) is a semantics-preserving transformation. Its inverse is *compound attribute disaggregation*.

Summary of Lesson 10

- In this lesson, we have studied the following notions:
 - *coexistence* constraint
 - *exclusive* constraint
 - *at-least-one* constraint
 - *exactly-one* constraint
 - *generic*, or user-defined, constraints
 - schema transformation, inverse transformation, reversible transformation

- We have also learned how
 - to define *coexistent*, *exclusive*, *at-least-one*, *exactly-one* groups:
 - in the Group box, click on the Coexistent, Exclusive, At-least-one button
 - to define a *generic* group constraint:
 - in the Group box, fill the User constraint field
 - to define a *generic* inter-group constraint:
 - in the Group box, click on the Constraint button
 - to define a compound attribute from its components:
 - if needed, make a group with the components; then
Transform / Group / Aggregate
 - to disaggregate a compound attribute:
 - Transform / Attribute / Disaggregate**

Exercises for Lesson 10

10.1 Let us consider the four schemas PERSONNEL that have been built in Exercise 9.3. For each of them, derive another schema in which the IS-A relation has been eliminated. Proceed as follows:

- replace the supertype/subtype relation by a one-to-one relationship type drawn between the supertype (cardinality [0-1]) and each subtype (cardinality [1-1]).

Take special care with all the derived existence constraints that express the subtype properties.

10.2 Let us consider the four schemas PERSONNEL that have been built in Exercise 9.3. For each of them, derive another schema in which the IS-A relation has been eliminated. Proceed as follows:

- propagate (by inheritance) all the components of the supertype (attributes, roles, constraints) to each of its subtype;
- remove the supertype.

Pay special attention to all the derived existence constraints. Be aware that employees who are neither clerks nor workers must be represented anyway.

10.3 Let us consider the four schemas PERSONNEL that have been built in Exercise 9.3. For each of them, derive another schema in which the IS-A relation has been eliminated. Proceed as follows:

- move all the properties of the subtypes to their supertype; for instance, the fact that all clerks have a function can be represented by an optional attribute of EMPLOYEE (see Figure 10.13);
- when all the properties have been pushed up to the supertype, remove the subtypes.

Take special care with all the derived integrity constraints. The role of an employee (clerk, worker, both or none) can be represented through, for example, new attribute `EmployeeType`.

10.4 Can you formulate an opinion concerning these three techniques to eliminate super-type/subtype relations? Some criteria: readability, sim-

plicity, conciseness, complexity of the additional integrity constraints, ease of translation into a relational database.

Do you think that some of these techniques are more fitted in some situations (think of subtype properties for instance)?

Note.

The problem of IS-A relation translation is complex, particularly when the database is to be implemented into a standard DBMS (e.g., a relational DBMS). It will be dealt with in a future lesson. Nevertheless, the techniques described in the questions above represent the three standard families of IS-A relation representations.

- 10.5 A relational database includes two tables, A and B, built by the following SQL program (column domains are ignored for simplicity):

```
create table A (A1 not null, A2 not null, A3, A4,
               primary key (A1,A2))

create table B (B1 not null, B2, B3, B4,
               primary key (B1),
               foreign key (B3,B4) references A)
```

Represent these structures by a logical schema.

Observe that the foreign key is optional. Ideally, two cases only are valid: either columns B3 and B4 both are null, or both have a value, in which case these values must match an A row. Represent this constraint in the logical schema.

Propose an equivalent conceptual schema.

- 10.6 Build entity type PERSON with, among others, optional attributes Country, Area, Local. Express the fact that these attributes are simultaneously *null* or *valued*. Make a compound attribute from them and call it Telephone.
- 10.7 Add to entity type PERSON mandatory attribute Address, made of (Number, Street and City); City is in turn a compound attribute comprising ZipCode and CityName.
- Disaggregate these attributes.
 - Make Address optional then apply the same manipulations.
 - Starting from these resulting flat structures, try to go back to the nested structures.

- 10.8 Consider once again the entity type PERSON. Add two entity types, namely COMPANY and ADMINISTRATION. A person can work in a company (where s/he receives a salary), in an administration (where s/he has a level) or is unemployed (in which case s/he receives an unemployment allowance). Add the necessary attributes and/or relationship types to represent these facts. **Without resorting to IS-A relations**, add the group constraints expressing the following situations:
- a person must either be in a company, or in an administration or unemployed, but only in one of these situations;
 - a person can either be in a company, or in an administration or unemployed, or nothing at all, but only in one of these situations;
 - a person must be in a company, or in an administration or unemployed, or in more than one of these situations;
 - a person can be in a company, or in an administration or unemployed, in more than one of these situations, or in none of them.
- Now, try to express these application domains **through IS-A relations**. Compare both expressions.
- 10.9 Design a schema that expresses the same idea as in Figure 10.21, but in which attribute Marriage is replaced by the entity type MARRIAGE.
- 10.10 Define a generic inter-group constraint that declares implication constraints.
- 10.11 Define a generic group constraint that states that the roles of a group must be played by different entities.
- 10.12 Define a generic inter-group constraint that allows designers to declare functional dependencies.

Lesson 11

More about relationship types

Objective

This lesson describes advanced constructs related with relationship types, namely *multi-ET roles* and *generic relationship types*. A multi-ET role can be played by an entity taken from one of several entity types. Instances of a generic rel-type are rel-types that can appear at different places of a schema with the same name and the same meaning.

11.1 Introduction

We will describe multi-ET roles that can be used to simplify some schemas, to make them more concise and more readable. Generic rel-types, i.e., rel-types with the same semantics that appear in the same schema, will be found in some specific application domains. We will examine two of them: aggregation and topological rel-types.

We start DB-MAIN and we create a new project called Lesson11.

11.2 Multi-ET roles

Each role of a rel-type is played by *one* entity that comes from *one* entity type. Sometimes, we would like to express the fact that the entity that plays this role can be **of type A or of type B**, depending on the situation. As an example, we consider a company in which pieces of equipment can be borrowed. The borrower can be either an individual or a service. We can model these facts by drawing a binary rel-types from EQUIPMENT to ... *both* EMPLOYEE *and* SERVICE (Figure 11.1).

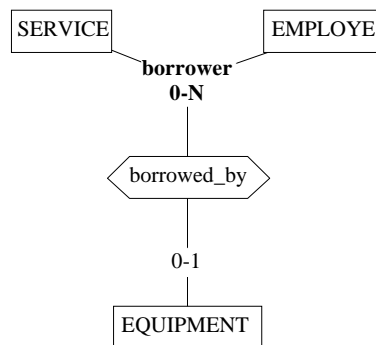




Figure 11.1 - The borrower of a piece of equipment is either a service or an employee.

Defining such a construct is easy:

1. draw `borrowed_by` rel-type from `SERVICE` and `EQUIPMENT` (with the button ); adjust the cardinalities if necessary;
2. draw a line from role `borrowed_by.SERVICE` and `EMPLOYEE` (with the button ); rename this role if necessary.

So, the role `borrower` is played by two entity types, hence its qualification: *multi-ET* role. Note that this does not mean that, in one particular instance of rel-type `borrowed_by`, this role can be played by several entities. Indeed, each instance comprises exactly two entities, one of type `EQUIPMENT` and one of type `SERVICE` or `EMPLOYEE`, just like in standard binary rel-types.

Now, would it have been possible to describe this part of the world in another way? As usual in database modeling, the answer is yes.

Let us first examine a tempting, but quite erroneous way to do it. It consists in defining a 3-ary rel-type involving entity types `SERVICE`, `EMPLOYEE` and `EQUIPMENT` (Figure 11.2). What is wrong with this schema? It tells us that a piece of equipment is borrowed simultaneously by **a service and an employee**, which obviously describes a quite different borrowing rule.

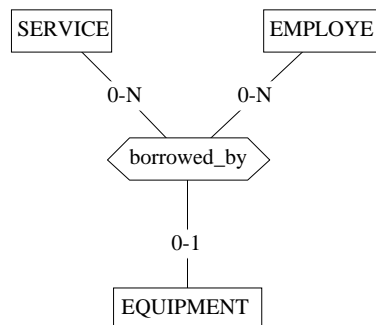


Figure 11.2 - This schema expresses a quite different situation from that of Figure 11.1!

On the contrary, Figure 11.3 tries (and succeeds!) to express the same meaning as Figure 11.1. In this schema, supertype `UNIT` generalizes all the organizational actors of the company, including services and employees, that can be responsible for borrowing pieces of equipment, and probably for other actions

as well. Therefore, a borrower is just a unit, which in turn is either a service or an employee.

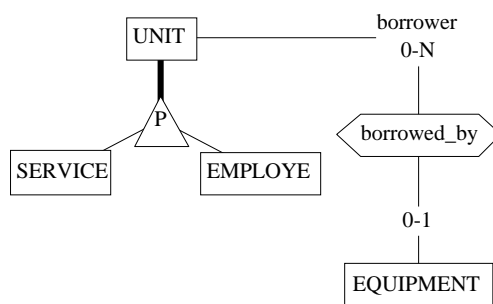


Figure 11.3 - Another way to describe the problem of Figure 11.1. The borrower of a piece of equipment is an **organizational unit**, which in turn is either a service or an employee.

The existence constraints studied in Lesson 10 give us still another way to model this situation. We can see things as follows:

The sentence

a piece of equipment is borrowed either by a service or by an employee;

can be rewritten without alteration of its meaning as

a piece of equipment either can be borrowed by a service or can be borrowed by an employee.

Hence the schema of Figure 11.4, which surely is far less elegant than the others (it includes two rel-types with the same semantics + a complex constraint), but which is quite correct too.

11.3 Generic rel-types

The rel-types defined in the schemas developed in the previous lessons represent *specific associations* between pairs (or tuples) of entities. Figure 5.15 and Figure 5.17 are typical examples of such rel-types. Defining other rel-types with the same name and the same semantics is fairly unlikely. However, some application domains may require similar rel-types to be defined in different

parts of the schema. We will discuss some examples and examine how to represent them.

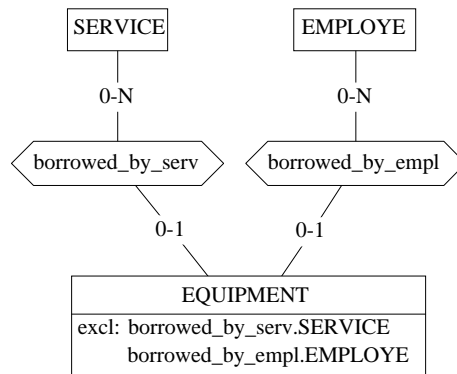


Figure 11.4 - Still another way to express multi-ET role of Figure 11.1.

Aggregation

Aggregation rel-types describe the composition of compound objects such as mechanical assemblies or chemical products. The schema of Figure 7.3 is the first example of such rel-types. It tells us that a car is perceived as the aggregation of 4 wheels, 1 body, 3 to 5 doors and 1 engine. Aggregation rel-types generally are called *part-of*, to tell that each component *is a part of* the compound object. The DB-MAIN model does not include a specific construct to represent aggregation rel-types, but such constructs can be represented in several ways, among which you can choose. We will use an example proposed in [Blaha, 1998] to illustrate these representations. The first schema makes use of specific rel-types that have distinct names (Figure 11.5). Such a schema offers poor readability because nothing suggests aggregation structures.

A much better representation mode consists in using generic *part-of* rel-types. All the rel-types in Figure 11.6 have a two-part name, consisting of the visible part *part-of* and the invisible part *|front-matter*, *|title-page*, *|toc*, etc. By dedicating the name *part-of* to aggregation structures, we can represent in a standard and readable way simple and complex aggregation structures. This technique produces a precise description of cardinality constraints: each *book* has one *front matter*, one or more *chapters* and, optionally, one *back matter*.

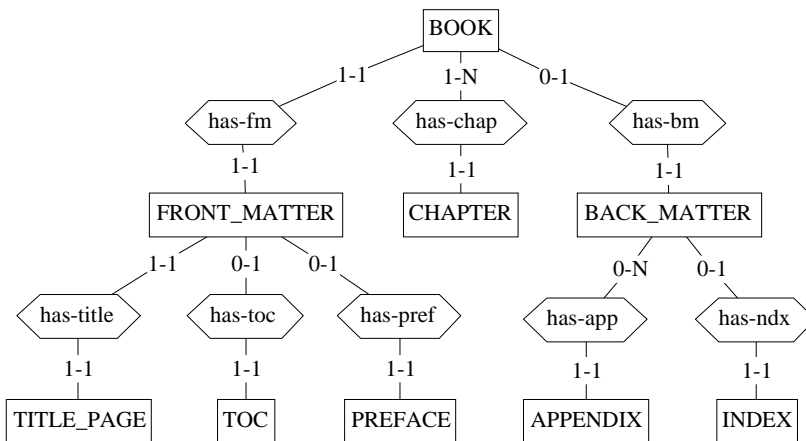


Figure 11.5 - Representing aggregation structures through standard rel-types.

If all the components of an assembly have the same cardinality, then an even simpler representation can be proposed through multi-ET roles (Figure 11.7, where the multi-ET role name has been set to "" to make it invisible).

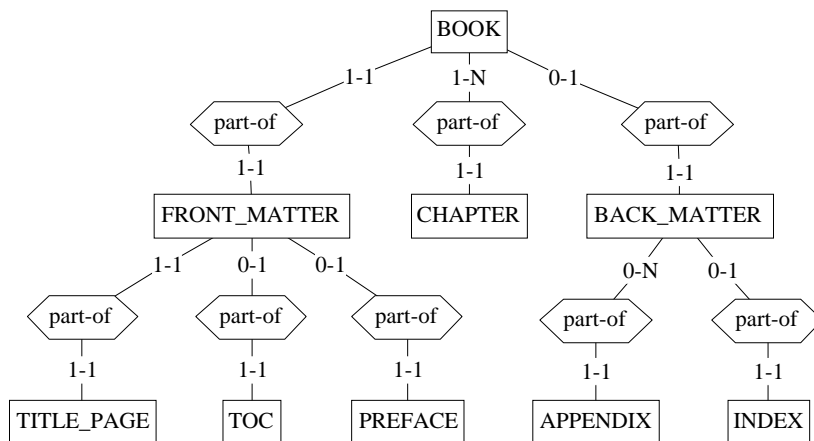


Figure 11.6 - Using the ambiguous name `part-of` to represent aggregation structures.

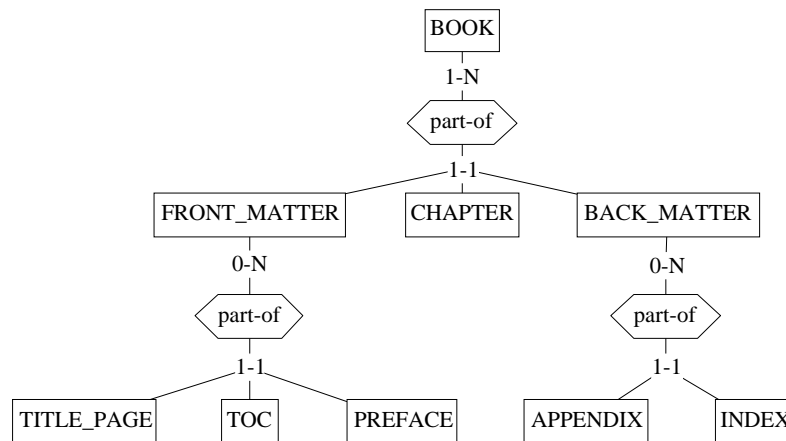


Figure 11.7 - The relation between a compound object and all its components is represented by **one rel-type** with ambiguous name and a multi-ET role.

Topological relationships

Spatial databases, such as those that underlie geographic information systems (GIS), record information about entities that have spatial properties, such as a form, a size and a position. Very often, users of spatial databases want to define topological relationships between objects such as *touch*, *cross*, *don't-cross*, *disjoined*. These relationships can have two roles. Firstly, that can be used as integrity constraints: *roads don't cross buildings*, *land parcels and lakes are disjoined*, *rivers cannot cross rivers*. Secondly, they can express interesting synthetic relations between entities: *buildings can touch roads*, *a river can touch another river*, *a road can cross roads*, *roads can cross rivers*, *buildings can touch buildings*. Of course, these relationships can be expressed as relations between the coordinates of the concerned spatial entities. However, such expressions are much less intuitive and expensive to compute (synthetic relations). The schema of Figure 11.8 describes a small geographical database in which generic relationship types express topological relationships.

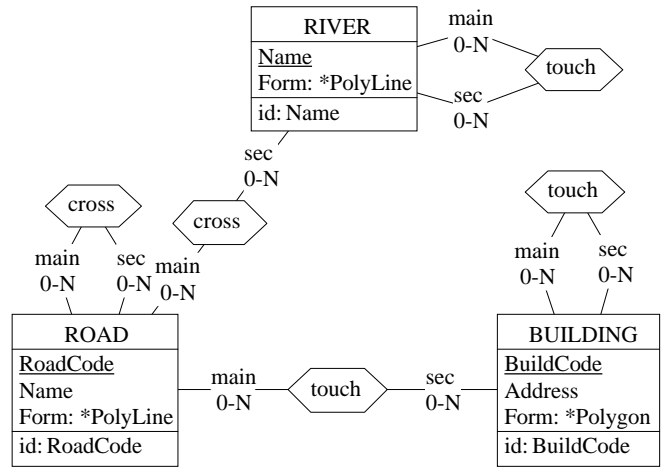


Figure 11.8 - Instances of two synthetic topological relations expressed as generic rel-types.

Key ideas of Lesson 11

1. Multi-ET roles

When a role is declared *multi-ET*, the entities that play this role can be of one of several types. This construct can also be expressed through a supertype or by duplicating the rel-type for each entity type that plays this role.

2. Generic rel-types

Some application domains require a given relationship type to appear in different places of schemas. Instead of defining as many standard rel-types, with different names, it can be better to use *generic rel-types*, that have the same name and the same meaning. Engineering and spatial databases intensively use generic rel-types.

Summary of Lesson 11

- In this lesson, we have studied the following notions:
 - multi-ET role
 - generic rel-type.

- We have also learned
 - to define a multi-ET role:
 - draw a single-ET role, then draw another line from this role to another ET.
 - to define a generic rel-type (reminder):
 - define a first instance of the rel-type, add the symbol "|" at the end of its name; create additional rel-types with the same name, followed with a distinct suffix (copy + paste works fine).

Exercises for Lesson 11

- 11.1 Propose a schema that describes the following situation with one rel-type:
services and employees can borrow computers and printers.
- 11.2 We consider the following application domain:
A meal comprises a first course, a main course and a dessert. A first course can be a cold dish, a warm dish or a soup. A main course can be a fish or a meat dish. A dessert can be pastry, ice cream or fruit.
Model this application domain by three equivalent schemas based on multi-ET roles, standard rel-types and IS-A relations.
- 11.3 Model with generic rel-types the structure of cars, seen as mechanical and electrical assemblies.
- 11.4 *Schemas comprise entity types and rel-types. A rel-type comprises roles, defined on entity types. Entity types and rel-types comprise attributes; the same can be said of compound attributes. Atomic attributes are defined on domains. Some domains comprise sub-domains.*
Model this application domain by using generic rel-types.
Note. The resulting schema describes schemas (including itself) and can be called a *meta-schema*. Each CASE tool includes a database in which the descriptions of schemas are recorded. Such a meta-database, whose schema is a meta-schema, is generally called *encyclopedia* or *repository*. The repositories of DB-MAIN are stored in *.1un files.

Lesson 12

View schemas

Objective

This lesson introduces a powerful feature of CASE tools, namely the concept of *view*. A view, or *view schema*, is a schema that includes a subset of a source schema. A view can be defined, generated and updated. The source objects themselves can be modified. These modifications can be propagated down to the view schemas.

Preliminary checking

We will use a new project, called Views. It includes the schema of Figure 12.1.

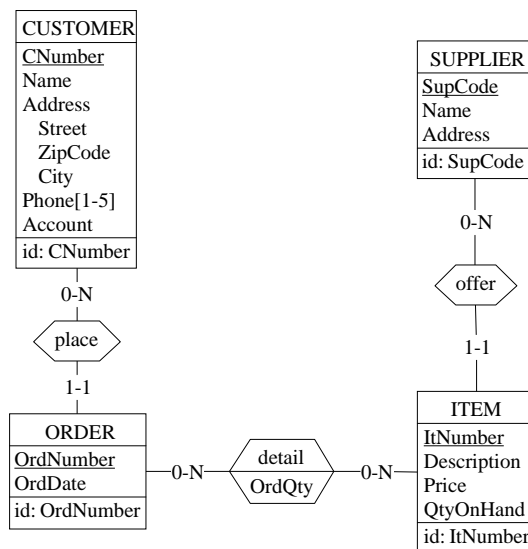


Figure 12.1 - The reference source schema.

12.1 Introduction

A DB-MAIN **view** is a particular presentation of a subset of a source schema. In its simplest form (called **latent view**), a view is just a named collection of objects belonging to the source schema. It can also be materialized as an explicit schema, called a **view schema**. A view schema includes all the objects of its corresponding latent view. You cannot *add*, *modify* or *delete* objects from a view schema. However, you can *rename* them, *transform* them or *move* them in the graphical space.

When to use views?

A view can be used to specify and display the part of a schema that describes a specific subset or a particular aspect of the application domain. Views are useful to decompose a large and complex schema into manageable subsche-



mas, making it more readable. A view will also be used to specify objects on which further operations have to be applied (such as checking, generating, reporting, etc).

Principles

In short, views work as follows:

1. The objects of a view are marked in the source schema.
2. Then, a view including the marked objects is defined by giving it a name. So far, the view is *latent*.
3. A latent view can also be materialized through the generation of a *view schema* that includes a copy of all the objects of the view. If an attribute is included, then its parent object (entity type, rel-type or compound attribute) is included as well. If an entity type, rel-type or compound attribute is marked, you can ask for their *attributes* and *processing units* to be included as well.
4. When modifications are applied on source schema objects, you can ask for refreshing the view schemas in order to propagate these modifications.
5. A *view schema* can be reworked: moving objects, renaming and transforming them. These modifications are preserved when you refresh the view schema.

12.2 Specifying the objects of the view

Open the only schema of the project. Choose a free marking plane; unmark the objects if needed (**Edit / Select marked**, then press the button , or **Edit / Select all**, then press the button  twice).

Select the desired objects and mark them (Figure 12.2):

- if you mark an *entity type*, a *rel-type* or a *compound attribute*, their attributes and groups can be asked to be automatically selected when generating the view schema;
- if you mark a *rel-type*, the roles whose entity types are selected are included in the view as well;
- if you mark an *attribute*, its parent will be implicitly included in the view;
- a *group* is implicitly selected if all its components are selected;

- if you mark a *collection*, only the marked entity types will appear;
- an *IS-A relation* is implicitly selected if the supertype and at least one subtype are selected;
- if you mark an entity type, a rel-type or a schema, their *processing units* can be asked to be inserted in the view schema.

In the example of Figure 12.2, all the marked objects will belong to the view, together with entity types CUSTOMER and ITEM, and attribute Address, since they are parents of selected objects. Figure 12.3 shows all the components that are explicitly and implicitly inserted in the view.

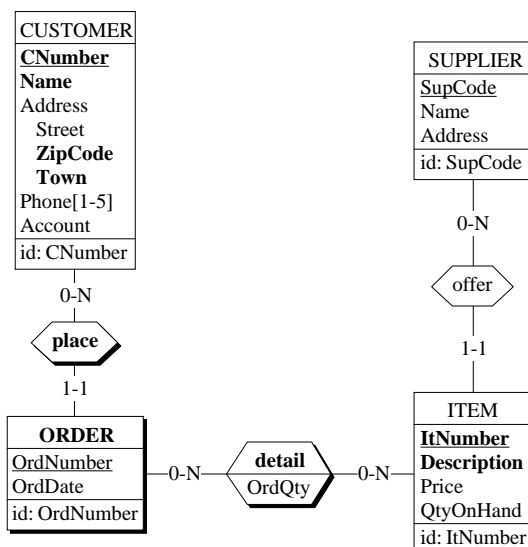


Figure 12.2 - The components to insert in the view are marked.

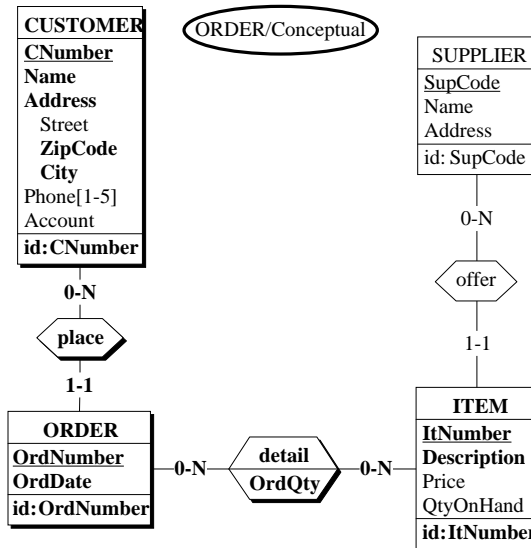


Figure 12.3 - The components that **actually** are included in the view, thanks to the propagation rules.

12.3 Defining the view

Now, we define a view comprising the marked objects. We execute the command **Product / View / Define view**. We call the view **Customers&Orders** (Figure 12.4).

12.4 Displaying a latent view

Later on, we can retrieve all the objects that make a latent view by the command **Product / View / Mark view**. The objects of the selected view appear to be marked. Note that this operation first cleans the current marking plane, and that former markings are lost.

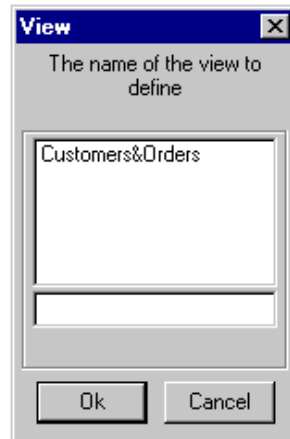


Figure 12.4 - Defining a view with the name Customers&Orders.

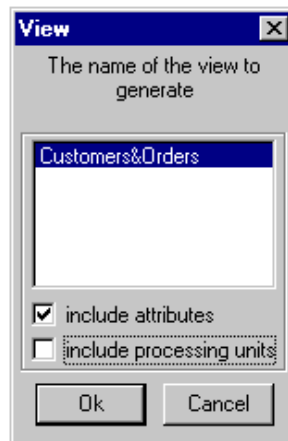


Figure 12.5 - Selecting a latent view to materialize it as a *view schema*.

12.5 Materializing a view as a view schema

Now, let's go for more interesting things. We want to create a new schema that comprises the objects of a latent view. We can call this *materializing* or *generating the latent view*. The new schema is a bit special, as we will see.

We execute command **Product / View / Generate view** (Figure 12.5). We select a view name (not much choice so far!). All the marked objects, as well as their parents, will be copied into the new schema.

However, we can tell DB-MAIN to include the components of marked objects by clicking on the include attributes button (**do it!**). In this case, all the attributes and groups of the selected entity types, rel-types and compound attributes are copied as well. Otherwise, only selected objects, together with their parents, are copied.

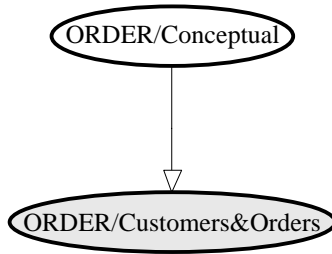


Figure 12.6 - A standard schema and a materialized view schema.

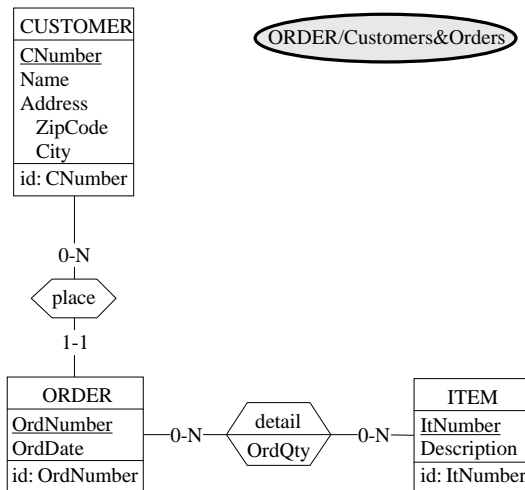


Figure 12.7 - The contents of view schema ORDER/Customers&Orders.

A new product is added to the project: *view schema* ORDER/CUSTOMERS&ORDERS (Figure 12.6). Opening this view shows the contents of the materialized view (Figure 12.7).

12.6 Modifying a view schema

Things can get more challenging when, as time goes on, we want to modify the view schema or its source schema. Let us first consider the first problem. Normally, a view is just a subset of its source schema. Consequently, adding or deleting entity types, rel-types, attributes, groups, or any other objects, in a view schema should be prohibited. However, some lighter, *cosmetic*, operations could be useful, and therefore allowed, provided they do not change the *semantics* of the objects.

For instance, the objects of the view schema, while quite comfortable in their natural environment (the source schema) can get an awkward and distorted look when they appear in the view schema. Therefore, moving and aligning the objects should be allowed. In addition, transforming objects changes their appearance, but not their meaning. So, transformations should be allowed either. Finally, just changing the name of an object does not change its very meaning, while improving its readability for given classes of users.

In summary, the objects of a view schema can be changed as follows:

- objects can be renamed;
- objects can be transformed;
- objects can be moved.

Other direct operations which may change the semantics of the view (add, delete, modify objects and their properties) are prohibited.

Figure 12.8 illustrates some possible changes that can be performed on our view schema:

- the entity type ITEM has been renamed PRODUCT;
- the rel-type detail has been transformed into the entity type REFERENCE;
- various objects have been moved.

Though their visual presentations are different, the schemas of Figure 12.7 and Figure 12.8 convey the same semantics, i.e., they represent the same portion of the application domain.

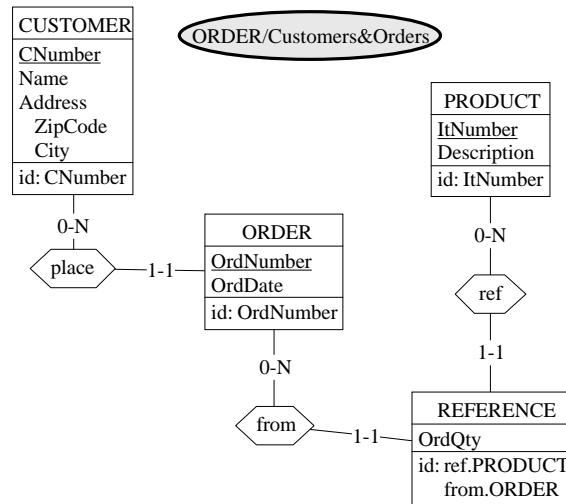


Figure 12.8 - View schema of Figure 12.7 manually updated.

12.7 What if I change my mind about the view?

Let us suppose that we want to **add objects to the view** and to **remove objects**. We just have to modify its definition, and to regenerate the new version.

Changing the latent view

- we open the source schema, and we select a free marking plane (needless to clean it);
- we display the latent view: **Product / View / Mark view**;
- we mark the new objects to insert and we unmark those to remove; for instance, we remove (unmark) CUSTOMER.Address.ZipCode and we add (mark) CUSTOMER.Address.Street and PRODUCT.Price;
- we redefine the view: **Product / View / Define view** in which we select view name Customers&Orders.

(Re)generating the view schema

- we just call **Product / View / Generate view** and select view name Customers&Orders.

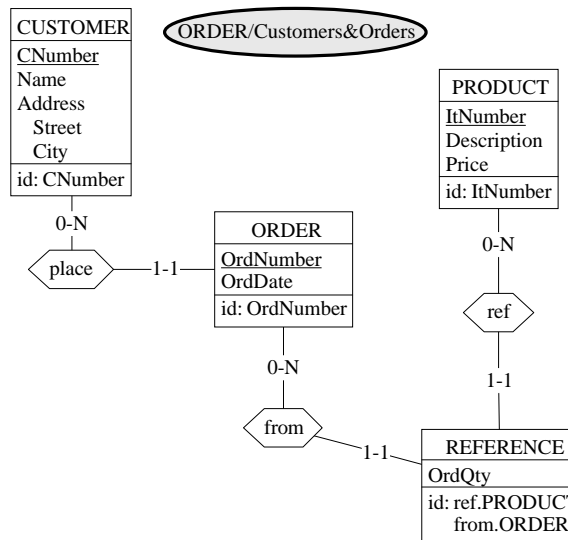


Figure 12.9 - The new version of view Customers&Orders.

Now, the new objects appear in the view schema while those discarded have been removed. In addition, *all the modifications made in the view schema have been applied* (Figure 12.9).

12.8 Modifying the source schema

So far, so good. But what if we want to **modify the source schema** itself? Schema changes are frequent, following the evolution of the application domain. If the view schema has been kept in its generation state, the situation is easy to master:

- we *modify the source schema*, adding, deleting and modifying objects;
- if needed, we modify the definition of the latent view,
- we *redefine the view*
- and we *regenerate it*.

The problem can be more complex *if we have reworked the view schema*, as in the case of Customers&Orders.

As an example of this situation, let us consider that we *modify the source schema* as follows (Figure 12.10).

- attribute City is renamed as Town
- new attribute DelivDate is added to ORDER
- cardinality [0-N] of detail.ORDER changed into [1-20]

Now the big question: what about the views of this schema? Obviously, they are obsolete and should be updated.

The first step clearly is to update the latent view by redefining it: **Product / View / Define view**.

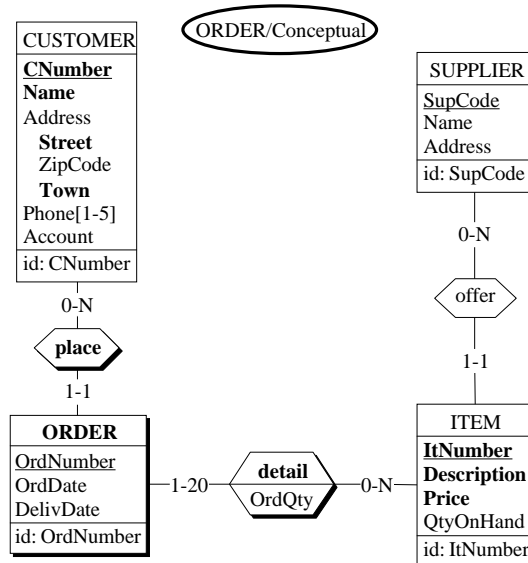


Figure 12.10 - The source schema has been modified. What about the views?

12.9 Propagating the modification of the source schema to view schemas

To propagate these modifications to the *view schema*, we have to rebuild it explicitly. We just have to generate the view again through **Product / View / Generate view**. The resulting view schema includes both the modifications of the

source schema and the reworking it was submitted to since its first materialization (Figure 12.11).

Note. The source of a view schema can be a view schema itself. However, in this case, the derived schema will be lost when refreshing the source view schema. This structure is adequate for static hierarchies of views.

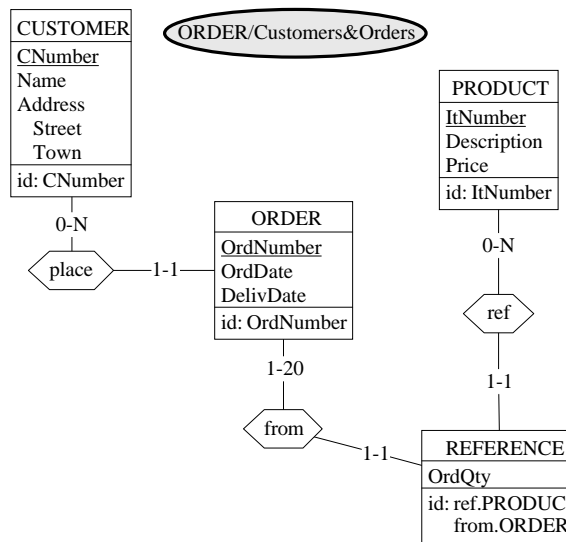


Figure 12.11 - The updated view schema including the new versions of the modified objects of the source schema.

12.10 Warning

Not all source schema modifications can be propagated to the view schema. Indeed, an object is known by its name. **If a source object is renamed or removed after it has been modified in the view schema, DB-MAIN is unable to process it when refreshing the view schema.** In case of renaming, the object in the view schema keeps its former state.

For example, if we change the name of rel-type detail into sub-order (Figure 12.12), DB-MAIN will be unable to cope with this rel-type (Figure 12.13), and all the view updating operations related to detail will be ignored (Figure 12.14).

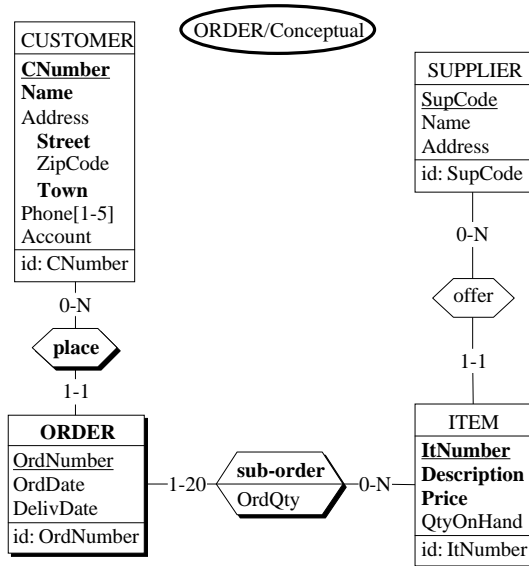


Figure 12.12 - Rel-type detail has been renamed as sub-order.



Figure 12.13 - DB-MAIN is unable to apply the view modifications on detail.

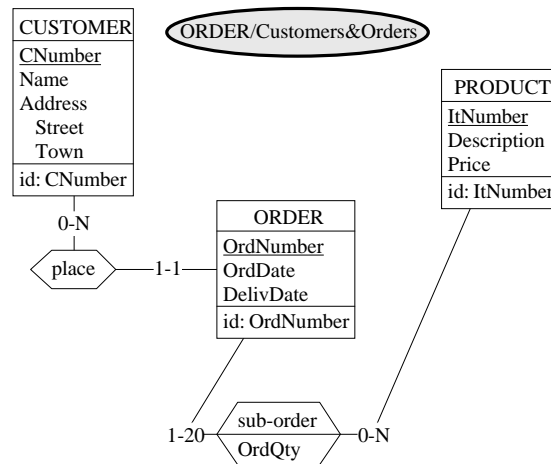


Figure 12.14 - The incompletely generated view schema.

12.11 Other operations

Other operations are proposed:

- *Remove view*: removes a latent view and its view schema, if any.
- *Copy view*: defines a new latent view, with a specified name, that includes the same objects as an existing view. No view schema is generated.
- *Rename view*: changes the name of a latent view and of the corresponding view schema, if any.

12.12 Technical information

In DB-MAIN versions 3 and 4, views are implemented through system dynamic properties:

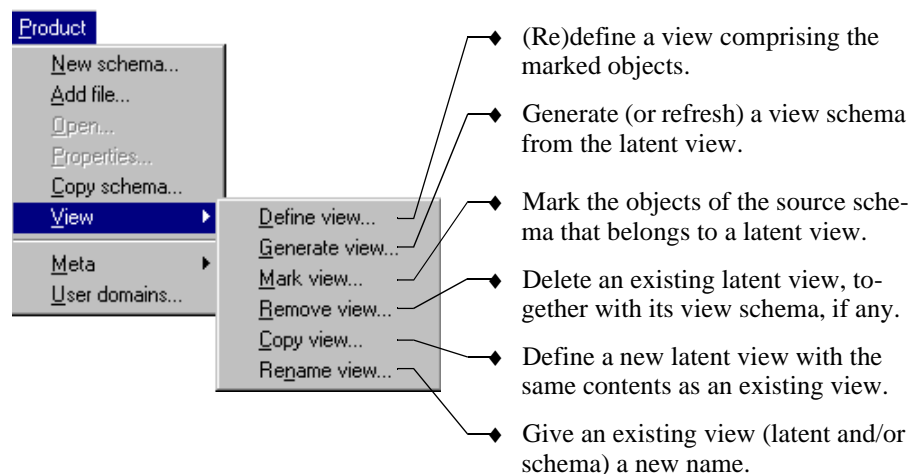
- **schema**: property list_view specifies the list of views it is the source of; property lsView indicates whether this schema is a view or not;
- **objects**: property view specifies the list of views of which this object is a component.

For each view schema, a **history journal** is automatically opened. This journal logs all the operations performed on the view schema. When refreshing a view, the tool automatically replays this journal. If a source object is renamed, the journal will ignore it when replayed since it knows it by its former name only. Similarly, if the name of an object is given to another object, indeterminacy problems may arise.

Note. Let us mention that there exists a *cheat tool* that allows us to change a *view schema* into a *plain schema* and conversely. Not recommended of course!

12.13 The View Menu

A short synthesis on how views can be managed:



12.14 There are *views* and *views*!

It goes without saying that the views we have discussed in this lesson have nothing to do with the various views (or display formats) of a schema we described in Lesson 2!

We could say, for instance, that *each view schema of a source schema can be examined according to each text or graphical view!*

Key ideas of Lesson 12

1. A **view of a source schema** is a schema whose semantics are a subset of that of the source schema.
2. Views of a large schema are a privileged way to define homogeneous parts corresponding to subsystems of the application domains.
3. A view can be latent or materialized. A **latent view** consists of a collection of marked objects in the source schema. A **materialized view** is a new product of the project, called a **view schema**, that includes the objects of a latent view + related objects defined by propagation rules.
4. A view schema can be **modified** through a limited set of operators: moving, transforming and renaming objects.
5. Modifications of source objects that are included in a view can be propagated to the view schemas (though with some restrictions).

Summary of Lesson 12

- In this lesson, we have studied:
 - the concept of view, and more precisely that of,
 - latent view
 - and materialized view, in the form of a view schema.

- We have also learned:
 - to define a latent view, through object marking, then,
Product / View / Define view
 - to generate a view schema
Product / View / Generate view
 - to mark the objects of a latent view
Product / View / Mark view
 - to remove a view from a source schema
Product / View / Remove view
 - to build a new latent view as a copy of an existing latent view
Product / View / Copy view
 - to rename a latent view and its view schema
Product / View / Rename view

- We have produced a new type of schema:
 - the view schemas.

Exercises for Lesson 12

12.1 Open the project `Library`. We consider three services of the library, namely,

- *Catalography*, concerned with books and their description;
- *Loan management*, concerned with borrowers to whom copies of books are loaned;
- *Borrower registration*, concerned with borrowers and the projects they borrow books for.

Define a view for each service. Materialize these rules into view schemas. Rearrange and transform the latter to give them a customized layout.

Change the source schema to practice the modification propagation rules.

Lesson 13

Text Processing

Objective

Texts have been largely overlooked so far. It is time to show that they can prove as important in database engineering activities as database schemas themselves.

We will first learn how to manipulate texts of any nature. Then, we will discuss the specific properties of computer-oriented texts, and briefly describe some processors that can extract essential information from them. In particular, we will examine a natural language extractor, DDL extractors, a text pattern analyzer, a dependency graph processor and a program slicer.

13.1 Introduction

A database project comprises schemas and text files. Though texts appear so far as mere output documents such as reports and SQL texts, they can prove a very rich information source for major engineering processes ranging from conceptual schema design to reverse engineering. We will first examine how to manipulate text files, then we will say some words about the structures that underlie text files.

13.2 Text file manipulation

In all the lessons of this tutorial, we have concentrated on database schemas. However, a project generally includes not only schemas, but also text files. We have somewhat overlooked these project components. There is not much to say about output text files such as the SQL programs or reports we produced in our small projects. They just have to be submitted to their favorite processor: a database engine for SQL files and a word processor for report files.

However, some projects may comprise input files in which important information can be extracted. We will mention three such file categories.


1. *Application domain documents.* When trying to model the concepts of a part of an application domain, we generally use various kind of information sources, the main of which are plain documents. Indeed, legal documents, accounting listing, sales reports, marketing brochures, interview transcripts, all include important information that can contribute to a better understanding of the application domain structure and behaviour.
2. *DDL files.* Very often, databases already exist, that implement a part of the application domain. Recovering the conceptual schema of existing databases (often called *legacy databases*), whatever their quality, is a major goal of the reverse engineering process. The first step is to build the physical or logical schema of these databases, an activity that is best carried out by analyzing their DDL text.
3. *Program files.* In the same context, analyzing old programs can bring us essential information on the structure, the properties and the management rules related to the files and databases of legacy applications.

Needless to say that these documents can be huge and complex, and that trying to analyze them by mere visual inspection can prove boring and highly unreliable. For instance, considering a large relational database of 500 tables¹,

6,000 columns, 1,000 indexes, 2,000 foreign keys, 800 triggers, and some additional technical adornments, the SQL script that encodes its structures can be more than 1,000 page long. Analyzing such a text will be particularly painful. Understanding a 30,000 line-of-code COBOL or PL/1 module which was written in the late seventies is not an easy task either².

Hence the need for specific text presentation and manipulation functions similar to those available for data structures in schemas.

To make the discussion more concrete, we create a new project and we include some text files in its workspace (if needed, we close the current project):

1. *Creating a new project.* We click on the New project button  (or execute the command **File / New project**).
2. *Including an external text file.* We search the DB-MAIN directory for a file named `library.txt`. We drag it from the Explorer window and drop it in the project window. Another way to include this file: execute the command **Product / Add text**.
3. *Including another external text file.* With the same procedure, we include a file named `library.ddl`.
4. *Including a third external text file.* ... and still another text called `order.cob`.

The project window looks like Figure 13.1.

We double-click on each of these products to examine their contents. The first one appears to be an interview report, the second one is a SQL script while the third one is a COBOL source text. Discussing specific processors for each of these text categories is beyond the scope of this volume. However, we will examine some of the main properties common to all texts files.

Selecting and marking text lines

We open the file `library.ddl`. We can make two interesting observations:

- Each line has a line number; however, this number is not part of the text, as we can observe by examining this text with a text processor³. Line numbers are used to reference specific parts of the text;

1. Databases of more than 1,000 tables are not uncommon.
2. Particularly if this module is just one component of a 3,000,000 LOC program.
3. If you have none available, you can use a standard text editor through the command **File / Edit text file**.

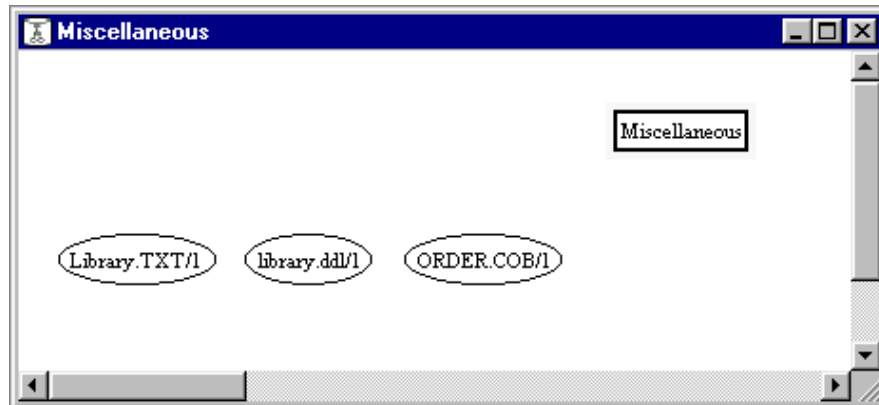


Figure 13.1 - Three text files have been included into the current project.

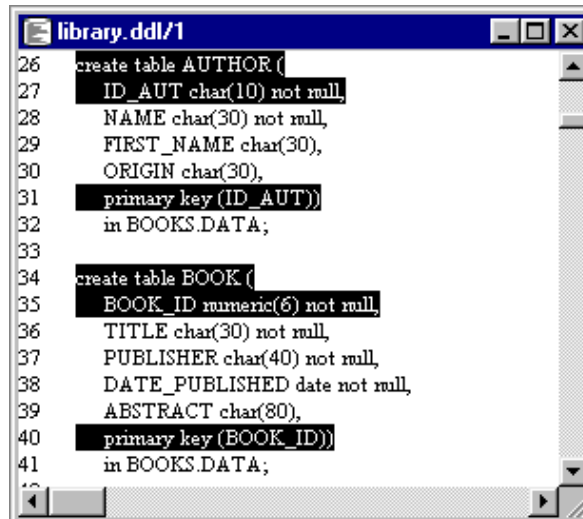
- The text cannot be modified: lines cannot be deleted, updated or inserted. Of course, the text can be changed through another text processor; however, this would not be a good idea, since then line numbers would no longer reference their target line.

Rule : do not add or delete lines in a text which currently is included in a project. Modifying an existing line can be harmless unless DB-MAIN has already analyzed the contents of this line. A text can be changed without problems, provided it has not yet been, or is no longer, included in a project.

Line selection. Clicking on a line selects it. Several lines can be simultaneously selected by using the shift and ctrl keys like in any Windows compliant application (Figure 13.2).

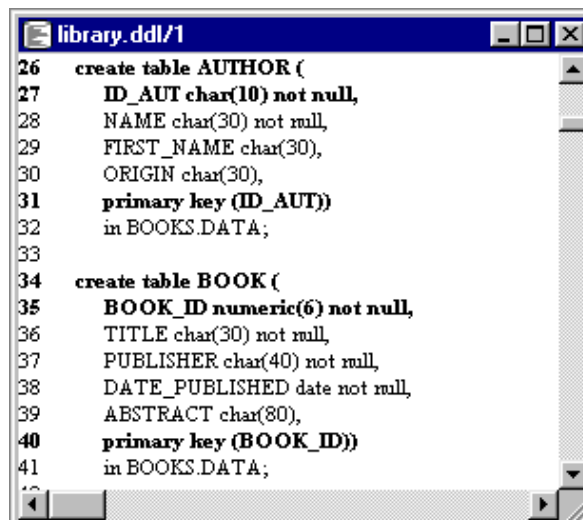
Line marking. Selected lines can be marked just like objects in a schema (Figure 13.3). Lines are marked in a definite marking plane through the button Mark. Up to five planes can be used, so that five different subsets of lines can be permanently maintained for each text file.

Line copy/paste. Selected lines can be copied on the clipboard (**Edit / Copy** or Ctrl+C), then pasted in a text of the project (SEM or TECH annotation for instance) or in any external text document. Line numbers can be included in the copy if requested: execute the command **Assist / Text analysis / Settings**, then check the button Copy line number.



```
26 create table AUTHOR (  
27     ID_AUT char(10) not null,  
28     NAME char(30) not null,  
29     FIRST_NAME char(30),  
30     ORIGIN char(30),  
31     primary key (ID_AUT)  
32     in BOOKS.DATA;  
33  
34 create table BOOK (  
35     BOOK_ID numeric(6) not null,  
36     TITLE char(30) not null,  
37     PUBLISHER char(40) not null,  
38     DATE_PUBLISHED date not null,  
39     ABSTRACT char(80),  
40     primary key (BOOK_ID)  
41     in BOOKS.DATA;
```

Figure 13.2 - Selected lines in a text file.



```
26 create table AUTHOR (  
27     ID_AUT char(10) not null,  
28     NAME char(30) not null,  
29     FIRST_NAME char(30),  
30     ORIGIN char(30),  
31     primary key (ID_AUT)  
32     in BOOKS.DATA;  
33  
34 create table BOOK (  
35     BOOK_ID numeric(6) not null,  
36     TITLE char(30) not null,  
37     PUBLISHER char(40) not null,  
38     DATE_PUBLISHED date not null,  
39     ABSTRACT char(80),  
40     primary key (BOOK_ID)  
41     in BOOKS.DATA;
```

Figure 13.3 - Marked lines in a text file.

Changing font, size and style. The font, as well as the character size and style, can be set through the command **Edit / Change font**.

Line annotation

By double-clicking on a text line, we open a text windows in which we can write comments, remarks or any kind of textual information (Figure 13.4).

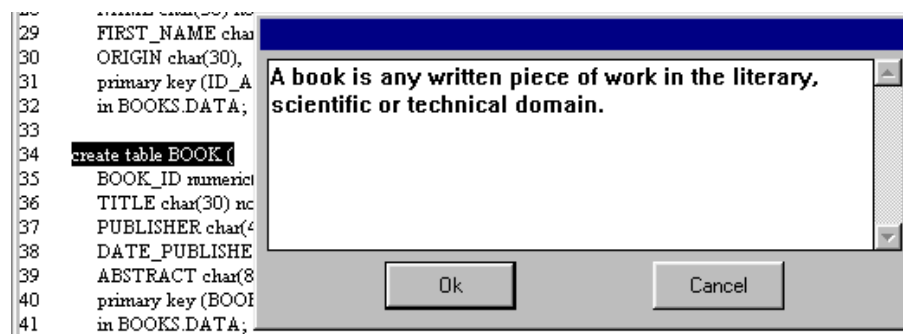


Figure 13.4 - Annotation associated with a text line.

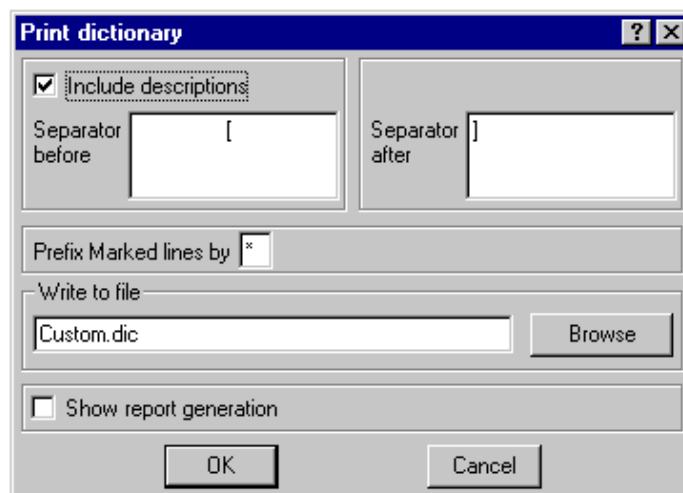


Figure 13.5 - Report definition box.

Reports from text files

A report can be produced from any text file (command **File / Print dictionary**). It is made up of all the lines of the source text, together with a marking symbol which specifies marked lines, and, if requested, with the text annotation associated with each line.

In the report defined in Figure 13.5, marked lines will be prefixed with symbol "*", and annotation will appear between a tab control + "[" and "]".

A fragment of the resulting report appears in Figure 13.6. The formatting rule has been set in the text processor in such a way that the annotations appear nicely outlined.

```
...
* create table AUTHOR ([Any author who participated in
                        the writing of a book recorded in
                        the library.]
*   ID_AUT char(10) not null,
    NAME char(30) not null,
    FIRST_NAME char(30),
    ORIGIN char(30),
*   primary key (ID_AUT))
    in BOOKS.DATA;

* create table BOOK ( [A book is any written piece of
                       work in the literary, scientific
                       or technical domain.]
*   BOOK_ID numeric(6) not null,
    TITLE char(30) not null,
    PUBLISHER char(40) not null,
    DATE_PUBLISHED date not null,
    ABSTRACT char(80),
*   primary key (BOOK_ID))
    in BOOKS.DATA;
...
```

Figure 13.6 - The resulting report with marking symbol and annotations.

13.3 Text structures and text analysis

So far, a text is just a string of characters, or at best, a sequence of lines.

In fact, many texts have a significant structure. Such is the case for DDL texts, which describe physical database schemas, and program source texts which define programming objects such as data types, variables and algorithms. Even plain texts in natural language can be considered as structured, provided they obey some definite grammar. Once the structure of a text can be recognized, useful abstractions can be extracted. For instance, a tool that knows the COBOL grammar can draw the paragraph calling tree and build the variable/statement cross-reference table from any COBOL program.

DB-MAIN includes a collection of tools devoted to text analysis and object extraction. We will briefly mention some of them. They will be detailed in another volume.

13.4 Natural language analysis

DB-MAIN includes a set of tools with which one can extract a conceptual schema from a simple text written in English⁴. The first component analyzes the text to check its grammatical correctness and to detect unknown verbs. The latter are classified, then introduced in a dictionary. The second component extracts the concepts and their relationships from the text to produce a first-cut conceptual schema. The third component normalizes the schema.

As an illustration, the English text of Figure 13.7 (top) has been analyzed and transformed into the schema of Figure 13.7 (bottom).

13.5 DDL physical schema extraction

DDL extractors are kinds of compilers that parse DDL texts in order to build the physical schema the text describes. To get an idea on what this means, try the following manipulations.

4. These tools are distributed in the DB-MAIN Application Library #2.

A book can be a literary-document, a scientific-document or a technical-document. A book is identified by a number. Each book is characterized by its title, the first-published-date, keywords, an abstract and its bibliographic-references. A book can have at most 6 keywords. A book is characterized by its physical-state. A technical-document must have a comment.

A book can be written by several authors. An author can have a first-name, a birth-date, and an origin. Each author has a surname. An author must write at least 1 book.

A book can be represented by several copies. The copies are identified by their ser-number. Each copy is characterized by its date and its location.

A copy can be borrowed by 1 borrower. Borrowers are identified by a personal-id. Borrowers are characterized by their name. Borrowers can have at most 5 phone-numbers. A borrower can borrow at most 5 books.

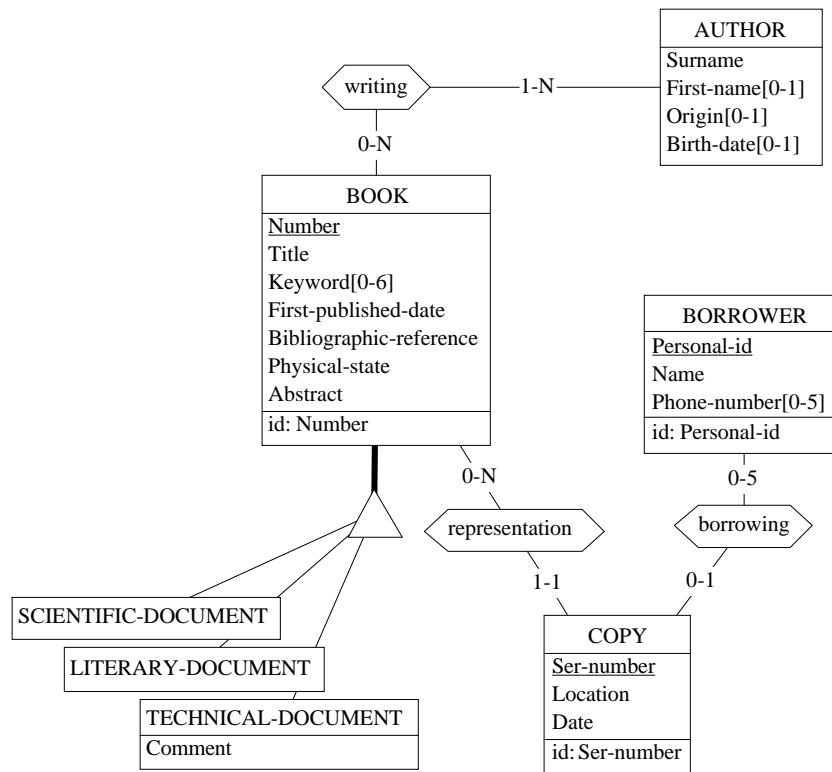


Figure 13.7 - The conceptual schema automatically extracted from the English text above.

- In the Project window, select text file `library.ddl/1`, then execute **File / Extract / SQL**. Examine the extraction report and the resulting schema.
- Now select text file `ORDER.COB/1` and execute **File / Extract / COBOL**. In the same way, examine the resulting schema.

This process is one of the first step of **database reverse engineering**. In most cases, more information must be extracted from various sources, such as views, application programs, screen and report layout, as well as from data. Furthermore, the completed logical schema must be *interpreted* or *conceptualized* that is, entity types, attributes, rel-types and various constraints must be built from these logical constructs. Though the SQL physical schema is fairly complete, and should be transformed into a conceptual schema without much problem, the COBOL schema is far too incomplete to lead to a decent conceptual schema. Further analysis of the procedural code of the program is needed to get additional knowledge on the record structures. More on this in another volume.

Note. If you are too eager to wait for another lesson, and you want to try recovering the conceptual schema, do as follows:

- execute the command **Assist / Global transformation**;
- click on the button **Predefined**, then select the script **Relational rev. eng.**
- click on the button **OK**.

So, you get a tentative conceptual schema. Unfortunately, things can be much more complex in actual systems. For instance, applying the same procedure on the COBOL physical schema is useless!v

13.6 Patterns

Texts which have a meaningful structure, such as any kind of programs, often include patterns. A *text pattern* is a formally defined text structure that can appear in several places in the text, and that is defined by a set of syntactic rules. Any section of text that satisfies these rules is a *instance* of this pattern.

For instance, a COBOL text file will include simple assignment statements which all look like:

```
MOVE <variable name> TO <variable name>
```

Text sections such as: "MOVE VAT-RATE TO A-FIELD" or "MOVE NAME OF RECA TO B" are two instances of this pattern.

Text patterns are defined as regular expressions expressed into a specific **pattern definition language** (PDL). The exact definition of the pattern above is as follows (see the *Text Analysis* Assistant):

```
cobol_name ::= /g"[a-zA-Z] [-a-zA-Z0-9]*";
cobol_var ::= cobol_name [- "OF" - cobol_name];
move ::= "MOVE" - cobol_var - "TO" - cobol_var ;
```

The first rule describes how COBOL variable names are formed (simplified): one letter possibly followed by a string made of dashes, letters and digits; letters can be in upper or lower case. The second rule defines two forms of variable designation: independent and component. The third rule expresses the basic form of the COBOL assignment statement.

Pattern analysis can be carried out through the *Text analysis* assistant.

13.7 Dependency graph

Useful abstract structures can be extracted from program files, such as dependency graphs. Program variable B is said to *depend on* variable A if the program includes an assignment statement such as "MOVE A TO B" or "B = A + C" or "LET B = SQRT(A)". The graph that describes all the variables together with the inter-dependencies is called the *dependency graph* of the program. As a general rule, the nature of the dependencies we are interested in are defined by the text patterns of the statements that generate them. DB-MAIN can build the dependency graph of a program, based on the definition of the patterns that define the dependencies. The user can then query the dependency graph by clicking on any variable in the source program.

Dependency graph building and querying can be done through the *Text analysis* assistant.

13.8 Program slice

When we consider a specific point (statement) S of a program P, we can be interested in collecting all the statements that will be executed just before the

program execution comes to this point. More precisely, we could ask to restrict these statements to only those which contribute to the state of a definite variable *V* used by *S*. This (hopefully small) sub-program *P'* is called the backward **slice** of *P* with respect to criterion (*S*; *V*).

Let us be more concrete, and consider statement 12,455 of the 30,000-line program *P*. This statement reads:

```
12455    WRITE COM INVALID KEY GOTO ERROR.
```

We want to understand which data have been stored into record *COM* before it is written on disk. All we want to know is in *P'*, the slice of *P* according to (12455 ; *COM*). *P'* is the minimum subset of the statements of *P* whose execution would give *COM* the same state as will give the execution of *P* in the same environment.

The goal of program slicing is obvious: trying to understand the properties of record *COM* is easier when examining a 200-line fragment than struggling with the complete 30,000-line program!

Text patterns, dependency graphs and program slices are very important concepts in program understanding activities, and therefore in database reverse engineering, which strongly relies on them. They all are available in the *Text analysis assistant*.

Key ideas of Lesson 13

1. A project includes schemas and texts. Input texts can include essential information to build new databases or to modify existing ones. Among the input texts we recognize three important classes, namely application domain texts, DDL texts that encode physical schemas and source programs.
2. Most text files are structured as a sequence of lines. Lines can be selected and marked in chosen marking planes. Each line can be given an annotation that includes various formal and informal information items such as comments.
3. Some application domain text files can be analyzed to find conceptual constructs.
4. DDL texts can be analyzed by language-specific extractors that build the physical schema that is encoded by this DDL file. Such a schema can then be conceptualized into its underlying conceptual schema.
5. Program files can be searched for specific *text patterns*. The *dependency graph* that defines the dependency relationships between variables can be built and queried. Excerpts of a program (*program slices*) can be derived by collecting the statements that contribute to the state of a variable at a given point of the program.
6. The program analysis techniques mentioned in this lesson will be developed in detail in another volume.

Summary of Lesson 13

- In this lesson, we have studied the following concepts:
 - text files, and more particularly input text files
 - projects and schemas

- We have mentioned (but not developed) sophisticated concepts and techniques that we will study later on:
 - DDL analysis
 - text patterns
 - dependency graphs
 - program slice

- We have also learned to:
 - include a text file in a project **Product / Add text** (or *drag&drop*)
 - select, mark, copy and paste text lines
 - change the font, the size or the style of a text file
 Edit / Change font
 - associate an annotation with a text line:
 double-click on the line
 - print a report from a text file: **File / Print dictionary**
 - extract a physical schema from a DDL file:
 File / Extract / <DDL brand>
 - conceptualize a physical schema: **Assist / Global transformation**
 then Predefined

Exercises for Lesson 13

- 13.1 Open the SQL text file `Library.ddl`. Mark all the table definition headers in plane 1, all column definitions in plane 2, primary key definitions in plane 3, foreign key definitions in plane 4 and index definitions in plane 5.

Create a new text in which you copy the header definitions only.

- 13.2 Take a copy of the file `Library.txt`. Split each line in such a way that each fragment (a bit modified and enriched if needed) now is represented by one object of the conceptual schema of the project `Library`. For example, the source sentences:

- *Every book has an identifying number, a title, a publisher, a first published*
- *date, key words, and an abstract (the abstracts are being encoded), the names*
- *of its authors, and its bibliographic references (i.e., the books it references).*

could be restructured into:

- *Every book has an identifying number*
- *Every book has a title*
- *Every book has a publisher*
- *Every book has a first published date*
- *Every book has key words*
- *Some book has an abstract (the abstracts are being encoded)*
- *Every book has the names of its authors*
- *Every book has its bibliographic references (i.e., the books it references).*

Now, copy each sentence in the semantic description (SEM) of the corresponding object.

- 13.3 Build a new project in which you include the SQL file `Manu-6.ddl` we generated in Lesson 6. Extract its physical schema. Conceptualize this schema with the script we used in Section 13.5. Compare the result with the source conceptual schema we developed in Lesson 5.

Lesson 16

Miscellaneous

Objective

This lesson discusses some additional functions of the DB-MAIN model and CASE tool that have not found their place in the other sections. First, two ways to extend the specification model are described and compared, namely semi-formal properties and dynamic properties. Both allow us to enrich the standard object classes (entity type, rel-type, attribute, etc.) with new, user-defined, properties.

Several parameters that govern the default behaviour of the tool are described. They allow users to customize their working environment.

16.1 Introduction

We will examine some features of the DB-MAIN model and tool that could make the developer's life easier, particularly in complex projects.

16.2 Generic properties

Object types (entity type, rel-type, attribute, etc.) have their own set of *built-in (or static) properties*. For instance, each attribute has a name, a short name, a cardinality, a collection type (optional), a type, stability and recyclability indicators, a length, a number of decimals (optional), a semantic annotation (button Sem) and a technical annotation (button Tech) (Figure 16.1). There is also a Prop(erty) button that will be described below.

The screenshot shows a dialog box titled "Attribute Properties" with the subtitle "Examine/modify the properties of an attribute of PRODUCT". The dialog contains the following fields and controls:

- Name: Text box containing "Supplier"
- Short name: Empty text box
- Cardinality: Dropdown menu showing "0-5" and a "Set" dropdown menu
- Type: Dropdown menu showing "Char", with checkboxes for "Stable" and "Non Recyclable" (both unchecked)
- Length: Spinner box showing "1"
- Buttons: "Sem.", "Tech.", "Prop.", "First att.", "Next att.", "Ok", and "Cancel"

Figure 16.1 - The built-in properties of attributes.

For some kinds of application domains, this set of properties could be considered as too poor to describe this domain adequately. On the other hand, it would be unrealistic to expect a general purpose CASE tool offering all the possible properties we could ever want. Therefore, a CASE tool should provide

a means to add new user defined object properties dynamically, what we can call *generic properties*.

DB-MAIN offers two kinds of generic properties, namely *semi-formal properties* and *dynamic properties*.

Though they are adequately managed by the tool (especially the dynamic properties), the latter is unaware of their meaning. Therefore, it cannot be asked to process them according to the intended semantics. Specific processing of generic properties must be developed as *Voyager 2* programs and procedures. This being a rather sophisticated point, it will be ignored in this tutorial. See the *Voyager 2* manual instead.

Semi-formal properties

Semantic and technical annotations are intended to associate free text descriptions with any specification object. For example, the semantic annotation of entity type PRODUCT will include a natural language definition of what we mean by "*a product*" in the application domain. In the technical annotation, we will rather specify some computer-oriented properties of the object, such as implementation mode or performance constraints.

However, it is possible to insert textual specifications in a more precise format in these annotations, namely the *semi-formal properties*. A semi-formal property is a new characteristics of the current object which is specified through the following statement:

```
#<property-name> = <property-value>
```

where <property-name> is the name of the property,
<property-value> is its value; the end mark of the value is either the end of the annotation or an *end-of-line* followed by the # character (these symbols being excluded).

In the example of Figure 16.2, the entity type PRODUCT has been given a semantic annotation which first gives a natural language specification of what is a product, then specifies five semi-formal properties

The qualifier "*semi-formal*" tells that (1) a precise syntax makes it possible to process (read, create, delete and update) such properties through specific *Voyager 2* procedures, but (2) there is no control on the consistency and the correctness of these specifications. Any typing mistake makes the sentence useless.

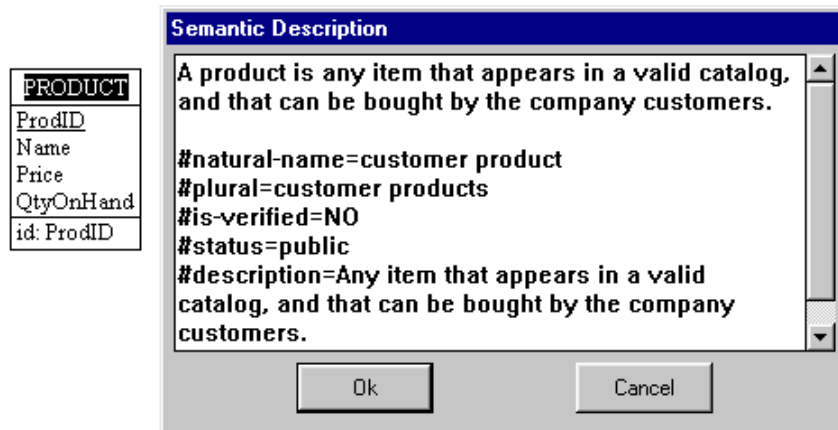


Figure 16.2 - Five semi-formal properties have been defined for the entity type PRODUCT. The last one, Description, has a multi-line value.

This technique is very flexible since it does not require changing the structure of the repository. However, it can prove unreliable for complex extensions. It will be particularly useful to build simple functional extensions of the tool.

Application. The natural language paraphraser¹ of DB-MAIN is a nice application of this technique. This processor generates a plain text description of a conceptual schema (in French in the current version; an English version is in preparation). Since the names of the objects may be *unnatural*, (e.g., QtyOnHand, ProID, ComAddress), and their gender as well as their plural form may be non standard, the user is asked to specify this information whenever standard rules do not apply. For instance, the (French) attribute name *Qte-Disponible* cannot be used as a correct noun in a text. We must tell the paraphraser that:

- the natural name is *quantité disponible*,
- the gender is feminine,
- the plural is *quantités disponibles*.

Hence the following semi-formal properties associated with this attribute:

```
#n=quantité disponible
```

1. Included in the Application Library #1.

```
#g=f  
#p=quantités disponibles
```

Defining a dynamic property surely is less flexible, but is a more structured and secure way to augment the modeling power of the tool.

Dynamic properties

A dynamic property is a characteristics associated with an object class of the repository in an explicit way. Adding such a characteristics must follow a strict procedure. First, it must be defined precisely (name, type, updatability, etc.), then only can it be used.

To help us grasp the concept, let us assume that we want to indicate, for each entity type, **which departments own the data** it describes. For instance, we would like to tell that the Personnel and Finance departments are the owners of the PRODUCT data. Obviously, such a property is unknown by the CASE tool. So we add it as follows.

Defining a dynamic property

We execute the command **Product / Meta / Properties**, which opens the dynamic property Management panel (Figure 16.3).

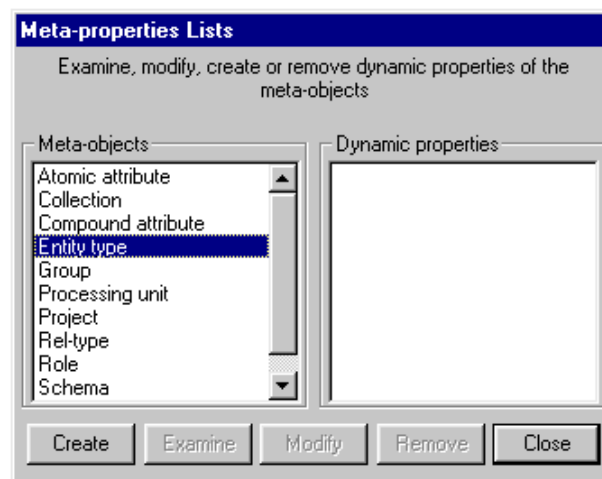


Figure 16.3 - The Dynamic property management panel.

We choose the object class to which we want the new property to be associated. In this case, we select the item Entity type. Then we click on the Create button.

The dynamic property definition box opens (Figure 16.4). We specify the name (Owner) and the type (string) of the dynamic property. We indicate that its values can be given and updated by the users (Updatable), that these values must be drawn from a predefined set (Predefined values) and that more than one value can be assigned (Multivalued).

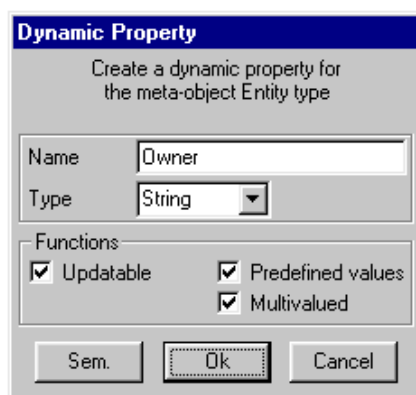


Figure 16.4 - Defining new dynamic property Owner as a list of character strings chosen from a predefined list.

It is good practice to associate a short description with a dynamic property, so that future users can understand its meaning. We click on the button Sem to open the desired box. We introduce this description, as well as the list of predefined values. For this, we follow the format of semi-formal properties (Figure 16.5). Finally, we confirm the operation (button OK).

Now, the dynamic property is defined and can be used to specify the owners of each entity type. Later on, we can update this definition, for instance by modifying the predefined value list.

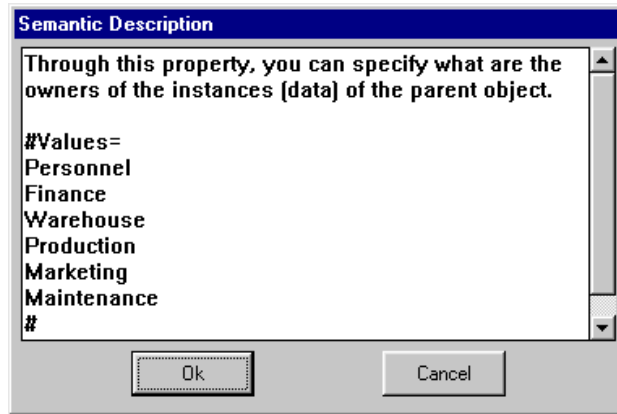


Figure 16.5 - Specifying the predefined values of the dynamic property Owner.

Setting and viewing dynamic property values

Now, we are ready to assign values to dynamic properties of each object. These properties are available from the property box of the object: we double-click on the entity type PRODUCT, then we click on the button Prop. An alternate way consists in merely selecting PRODUCT, then clicking on the PROP button in the Standard tools bar (Figure 16.6).

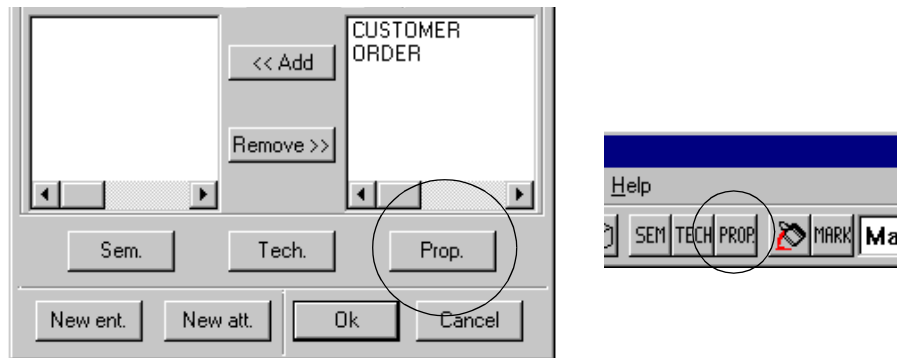


Figure 16.6 - How to get the Dynamic property box of an object: click on the Prop. button in the Property box of the object or on the PROP button in the Standard tools bar.

We get the dynamic property box of PRODUCT (Figure 16.7). In the left side list, we select the property Owner. Then, we select the values in the right side list, and we click on the button <<Add first. We confirm this choice (button OK). That's all.

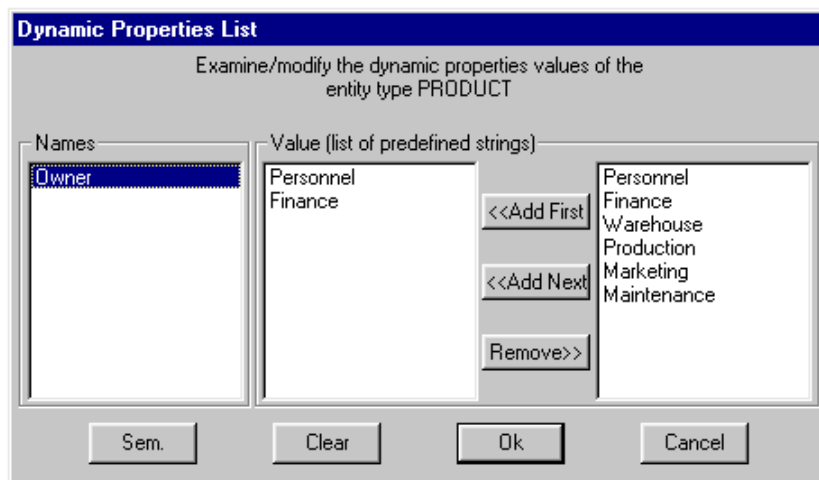


Figure 16.7 - Declaring that the departments Personnel and Finance are the owners of entity type PRODUCT.

If we only want to consult the properties of a series of objects, the simplest procedure is to open the general Property box through the command **Windows / Property box** (Figure 16.8).

So, what to do with dynamic properties? The answer is the same as for the semi-formal properties. They can be used to record more precise specifications. However, they show all their power when associated with specific procedures written in *Voyager 2*. It is important to note that some assistants can use dynamic properties as well, namely the *Schema analysis* and *Advanced global transformation* assistants. More on this in another volume.

Application. An add-on has been developed to allow data administrators to manage not only the corporate information, but also the organizational units, and their roles in information management².

-
2. This package, called ORGA, is included in Application Library #1. An example of the organizational unit map is shown in the *Function overview*.

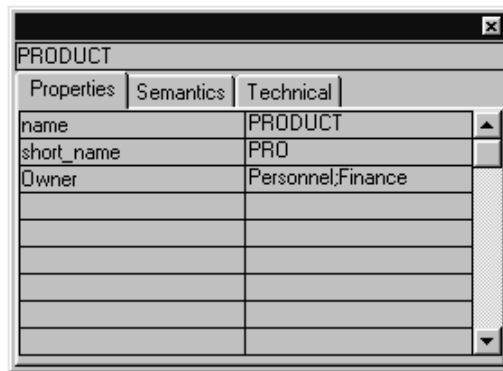


Figure 16.8 - The property box of an object shows all the properties, be they built-in or dynamic.

The system comprises three modules. The first one allows administrators to graphically define and update the hierarchical structure of the organization and the standard roles (owner, main user, responsible, security, validation, etc.). The second module introduces the roles and the organizational units in a target project. The third module provides easy procedures to assign units to information types according to definite roles and generate various reports. Each role appears as a dynamic property created by the second module. Its predefined values are the names of the organizational units. All the modules have been developed in *Voyager 2*.

16.3 Configuration settings

The configuration comprises the current values of the parameters of the DB-MAIN environment. These parameters represent your preferences as far as the behavior of the tool is concerned. They are independent of the projects.

The *Configuration Management panel* is opened by the command **File / Configuration** (Figure 16.9). The settings are saved in the file `c:\windows\db_main.ini`. Since the number and the nature of the parameters quickly evolve, your version will probably includes parameters different from those described below. At the present time, it allows users to define custom settings for the following parameters:

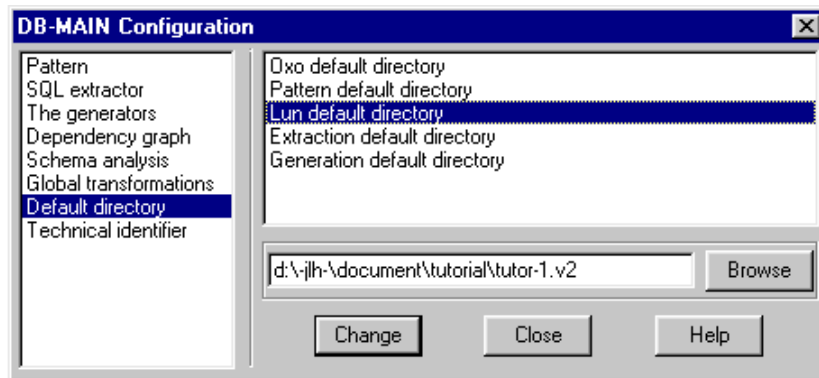


Figure 16.9 - The Configuration Management panel.

Text patterns. Concerns the *Text pattern matching* engine (**Assist / Text analysis / Search**). Specifies the names of the default main and secondary pattern libraries.

SQL extractor. Concerns the *SQL extractor* (**File / Extract / SQL**). Defines some settings such as: *Are the columns NULL by default? Should extracted views be stored in the same schema as the base tables?* Etc.

Code generators. Concerns the *DDL generators* (**File / Generate**). Specifies the name of specific generators such as COBOL and CODASYL DBTG. These generators are *Voyager 2* programs.

Dependency graph. Concerns the *Dependency graph* processor (**Assist / Text analysis / Dependency**). Specifies for instance whether isolated variables (participating in no relations) can be queried or not. If they can, selecting such a variable colors all its instances in the program (see the corresponding volume).

Schema analysis. Concerns the *Schema analysis* and *Advanced global transformation* assistants (in the **Assist** menu). Specifies the name of the default library of rules (file * .an1).

Global transformations. Concerns the *Advanced global transformation* assistants (in the **Assist** menu). Specifies the name of the default library of transformations (file * .tfl).

Default directories. Specifies the default directory for the main file types of DB-MAIN: *Voyager 2* programs (* .oxo), text pattern libraries (* .pat),

projects (*.lun), program files to be processed by extractors, DDL text generated by the generators, etc. (Figure 16.9)

Technical identifier. Defines the default type and length of technical identifiers. Used in the transformation **Transform / Entity type / Add Tech. id**

Key ideas of Lesson 16

1. Each object class, which any project is made up of, has static (built-in) properties such as *name*, *short name*, *type*, *length* and *semantic annotation*. CASE tools must provide users with means to add new properties. The DB-MAIN model offers two kinds of additional properties, namely semi-formal and dynamic properties.
2. A semi-formal property is defined as an expression in an annotation. This definition specifies the name and the value of the property. Such properties can be processed by *Voyager 2* procedures only.
3. A dynamic property is a declared property that is part of the repository. It must be defined before being used, and is structurally attached to an object class. Such a property can be processed by the global transformation and schema analysis assistants, as well as by *Voyager 2* procedures.
4. A CASE tool can be customized through a series of parameters that define its default behavior.

Summary of Lesson 16

- In this lesson, we have studied some important concepts:
 - the concept of semi-formal property
 - the concept of dynamic property

- We have also learned to:
 - define a new semi-formal property (in textual annotations)
 - define a dynamic property: **Product / Meta / Property**
 - examine and use a dynamic property:
 - buttons Prop and PROP
 - Windows / Property box**
 - change the configuration parameters:
 - File / Configuration**

- We have learned about a new file:
 - the DB-MAIN configuration file `db_main.ini`

Exercises for Lesson 16

- 16.1 Define a set of dynamic properties that allows developers to describe the level of confidence of the entity types of a schema.
- 16.2 Define a set of dynamic properties that allows developers to define the implementation mode of access keys (B-tree, hashing, etc.), as well as the page size and buffer size of each collection.
- 16.3 Change the default project directory.

Appendix A

The Generic DB-MAIN Model

Objective

The DB-MAIN tool allows analysts and developers to represent and specify information structures, data structures and processing units that make up an information system.

These specifications must comply with the so-called DB-MAIN specification model which defines the valid objects and their relationships. This appendix describes the main components and features of this model.

A.1 The specification model in short

The model includes a very small number of concepts: *projects*, *products* (schemas, views and text files), *entity types*, *relationship types*, *attributes*, *domains*, *groups*, *inter-group constraints*, *collections* and *processing units*.

However, due to their generality, these concepts can be used to describe in a precise way information systems at different levels of abstraction (conceptual, logical, physical) and according to various abstract or concrete paradigms: Entity-relationship, Object-role, Object-oriented, standard files, CODASYL DB-TG, IMS, TOTAL/IMAGE, relational, object-relational, etc.

Its ability to specify constructs at different level of abstraction and paradigms gives users a great level of flexibility that will prove useful for large projects in which several DBMS are used. It will also be most necessary in reverse engineering activities, where unfinished schemas frequently include physical, logical and conceptual constructs.

Specifications comprise two kinds of information, namely products and histories. We will describe the concepts of which products are made up, leaving the discussion of histories to another document.¹

A.2 Project

The highest level object is the project. It comprises all the specifications related to an engineering *project* as well as the history of all the activities that were carried out to produce these specifications.



Figure A.1 - Iconic representation of a project. Appears in the Project window.

A project is made of one or several products - or documents - which fall into two classes: **schemas** and **text files**. Each repository describes a project. It is

-
1. Be sure that the DB-MAIN tool is in the **View / Graph. dependency** mode to hide the histories. To avoid recording these histories, the **Trace** item of menu **Log** must be unchecked as well.

stored in a *.LUN file. A project can be entered manually by the user or can be imported from an *.ISL ASCII text file. Though it is easy to transfer specifications between projects (through the export-import functions), there is no explicit relation between two projects.

A.3 Base Schema

A *schema* is a complete or partial description of data structures and application processes (such as those found in files, in programs or in databases). A base schema can be built from scratch, can derive from another schema (e.g., through import, copy, integration or transformation) called its origin or can derive from an external text file, e.g., an SQL or CODASYL source file. A schema mainly consists of **entity types** (or *object classes*), **relationship types** (*rel-types* from now on) and **collections**. A schema can have **processing units**.

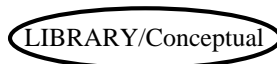


Figure A.2 - Iconic representation of a base schema. Appears in the Project and Schema windows.

A.4 View Schema

A *view schema* (or simply *view*) is a schema that derives from another schema S, called its source, and that includes a subset of the constructs of S. The constructs of a view can be renamed, transformed and moved in the graphical space, but no objects can be added or deleted.

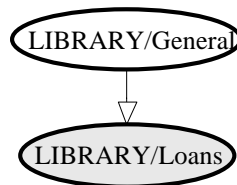


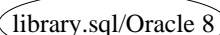
Figure A.3 - Iconic representation of a view schema and of its source schema. Appears in the Project and Schema windows.

Any update in the source schema *S* can be propagated down to the views that have been derived from it. A view can be derived from another view.

A view must first be defined as a *latent view*, which is a named subset of the source schema. A view schema materializes a latent view, from which it is generated.

A.5 Text file

A *text file* is an external text that generally, either derives from a schema (e.g., a generated SQL script file), or from which a schema has been (or will be) derived (e.g., a COBOL source text or an interview report). Text files are known, and can be processed by the tool, but their contents are not stored in the repository.



library.sql/Oracle 8

Figure A.4 - Iconic representation of a text file. Appears in the Project window.

A.6 Inter-product relationship

The products of a project, i.e., its schemas and its text files, can be linked by derivation relationships that express the way products are developed from other products (Figure A.5). These derivation relationships can be explicitly described through a hierarchy of processes (ignored in this appendix).

A.7 Entity type (or object class)

An entity type represents a class of concrete or abstract real-world entities, such as customers, orders, books, cars and accidents. It can also be used to model more computer-oriented constructs such as record types, tables, segments, and the like. This interpretation depends on the abstraction level of the schema and on the modeling paradigm in use. For instance, in an object-oriented model, we will use the term *object class* instead. Object classes generally are given methods and appear in IS-A hierarchies.

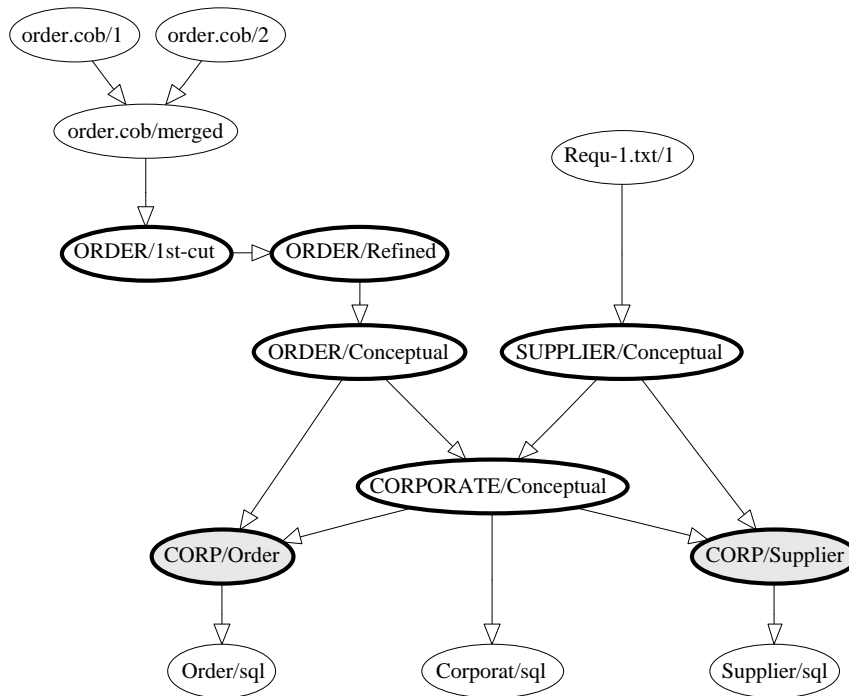


Figure A.5 - The network of products of a project. Includes base schemas, view schemas, input text files and output text files².

An entity type can be a subtype of one or several other entity types, called its supertypes. If F is a subtype of E, then each F entity is an E entity as well.

The collection of the subtypes of an entity type E is declared **total** (symbol **T**) if each E entity belongs to at least one subtype; otherwise, it is said to be partial.

This collection is declared **disjoint** (symbol **D**) if an entity of a subtype cannot belong to another subtype of E; otherwise, it is said to overlap. If this collection is both total and disjoint, it forms a **partition** (symbol **P**).

2. This display is obtained through the **dependency view** of the history (**View / Graph. dependency**).

An entity type can comprise **attributes**, can play roles in **rel-types**, can be collected into **collections**, can be given **constraints** (through **groups**) and can have **processing units**.

Since a supertype/subtype relation is interpreted as "each F entity *is a* E entity", it is called an **IS-A relation**. IS-A relations form what is called an **IS-A hierarchy**. Indeed, an entity type cannot be, directly or not, a subtype of itself.

An entity type can have more than one supertype. Such a situation is called multiple IS-A hierarchy, or more commonly (though improperly) multiple inheritance.

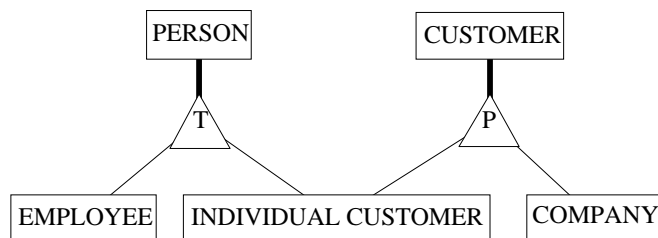
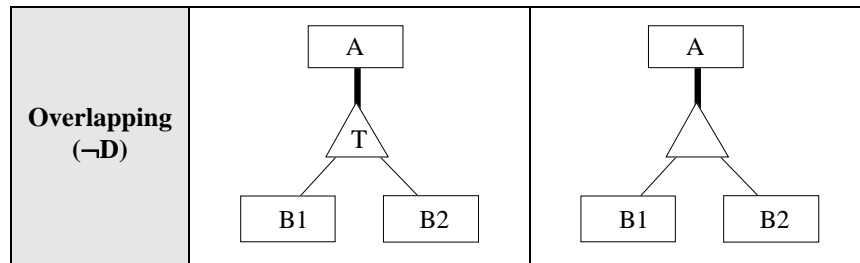


Figure A.6 - A hierarchy of entity types. PERSON and CUSTOMER are supertypes, EMPLOYEE, INDIVIDUAL CUSTOMER and COMPANY are subtypes.

The four supertype/subtype patterns are summarised in the table below, where B1 and B2 are two subtypes of A:

	Total (T)	Partial ($\neg T$)
Overlapping ($\neg D$)		



A.8 Relationship type (rel-type)

A *relationship type* represents a class of associations between entities. It consists of entity types, each playing a specific **role**. A rel-type with 2 roles is called **binary**, while a rel-type with more than 2 roles is generally called **N-ary**, where **N** is the **degree** of the rel-type. A rel-type with at least 2 roles taken by the same entity type is called **cyclic**.

Normally, a role is played by one entity type only. However, it can be played by more than one entity type. In this case, it is called a **multi-ET** role. In any relationship, this role is taken by an entity of one of these types.

Each role is characterized by its **cardinality** [*i* - *j*], a constraint stating that any entity of this type must appear, in *i* to *j* associations or relationships³. Generally *i* is 0 or 1, while *j* is 1 or *N* (= *many* or *infinity*). However, any pair of integers can be used, provided that $i \leq j$, $i \geq 0$ and $j > 0$.

Let us consider a binary rel-type *R* between *A* and *B* with cardinality [*i*_a - *j*_a] for *A*, [*i*_b - *j*_b] for *B*.



3. The reader must be aware that other interpretations of role cardinalities exist. In [Teorey,1998], [Elmasri,1994] and [Rumbaugh,1991], for instance, the cardinality of a role states how many relationships can/must exist for any combination of instances of the other roles. This interpretation is convenient for binary rel-types, but poses several problems for N-ary rel-types (see UML for instance). The current model is compliant with the interpretation of [Batini,1992], [Bodart,1994], [Nanci,1996] and [Coad, 1995].

R is called:

<i>one-to-one</i>	if $j_a = j_b = 1$
<i>one-to-many from A to B</i>	if $j_a > 1$ and $j_b = 1$
<i>many-to-one from A to B</i>	if $j_a = 1$ and $j_b > 1$
<i>many-to-many</i>	if $j_a > 1$ and $j_b > 1$
<i>optional for A</i>	if $i_a = 0$
<i>mandatory for A</i>	if $i_a > 0$.

A role can be given a name. When no explicit name is assigned, an implicit default name is assumed, namely the name of the participating entity type. The roles of a rel-type have distinct names, be they explicit or implicit. For instance, in a cyclic rel-type, at least one role must be given an explicit name. A multi-ET role must have an explicit name as well.

A rel-type can have **attributes**, and can be given **constraints** (through **groups**) and **processing units**.

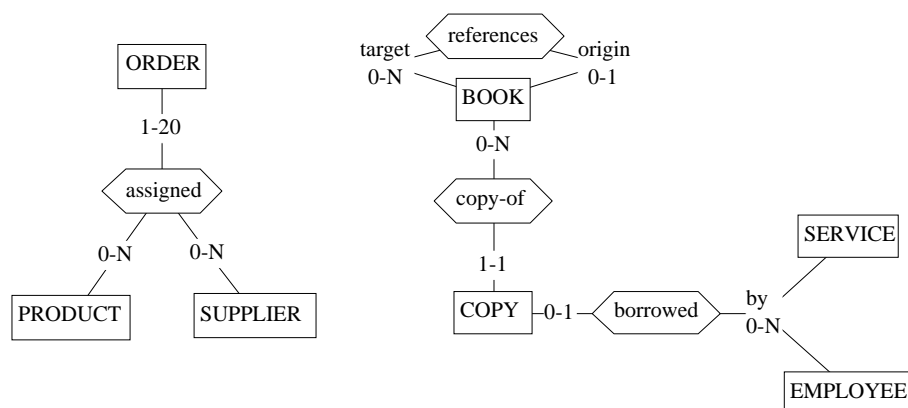


Figure A.7 - Relationship types. Rel-types references, copy-of and borrowed are binary, while assigned is 3-ary. Rel-type references is cyclic. Role borrowed.by is multi-ET. Copy-of and borrowed are functional. references is many-to-many.

The term *many role* designates a role with $j > 1$ and *one role* designates a role with $j = 1$. A one-to-many rel-type has 1 *many role* and 1 *one role*.

A rel-type which has attributes, or which is N-ary, will be called a **complex** rel-type. A one-to-one or one-to-many rel-type without attributes will be cal-

led **functional**, since it materialises a functional relation, in the mathematical sense.

A.9 Collection

A *collection* is an abstract or concrete repository for entities. A collection can comprise entities from different entity types, and the entities of a given type can be stored in several collections. Though this concept can be given different interpretations at different level of abstraction, it will most often be used in logical and physical schemas to represent files, data stores, table spaces, etc.

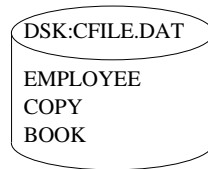


Figure A.8 - DSK:CFILE.DAT is a collection in which EMPLOYEE, COPY and BOOK entities can be stored.

A.10 Attribute

An attribute represents a common property of all the entities (or relationships) of a given type. Simple attributes have a **value domain** defined as a built-in data type (number, character, boolean, date, ...) and a length (1, 2, ..., 200, ..., N [standing for *infinity*]). These attributes are called **atomic**. The built-in domains have the following characteristics.

1. *Boolean(n)*: Set {true, false}, or any set of 2 elements.
2. *Char(n)*: The set of **n**-character strings.
3. *Varchar(n)*: The set of strings with length from 0 to **n**.
4. *Date(n)*: Set of dates or timestamps.
5. *Numeric(n[,d])*: Set of numerical values of **n** decimal digits, including **d** decimals.
6. *Float(n)*: Set of floating point numerical values with a representation of **n** bytes.

7. *Index(n)*: Numerical values that designate the elements of parent attribute A[I-J], which is a multivalued attribute of type *array*. If A has actual cardinality k, the index attribute instances takes **some** values from 1 to k.
8. *Sequence(n)*: Numerical values that designate the elements of parent attribute A, which is a multivalued attribute of type *list*. If A has actual cardinality k, the index attribute instances takes **all** the values from 1 to k.

In these definitions, **n** stands for the length (*Boolean, Char, Date, Numeric, Float, Index, Sequence*), or the max length (*Varchar*), of the domain values. For each type, the tool proposes a default length. Except for *Date* and *Boolean*, it is an unusual value that should, in most cases, be replaced. The rules for **n** are summarized in the following table.

Type	range of n	default	particular rule
Boolean	1-99	1	
Char	1-99999	1	
Varchar	1-99999; N	1	N stands for <i>unlimited length</i>
Date	1-99	10	
Numeric	1-99; 0-99	1	1st figure = total length 2nd figure = decimals
Float	1-99	1	
Index	1-9	1	not shorter than length of max card. of the array (e.g., 3 for max card. = 500)
Sequence	1-9	1	not shorter than length of max card. of the list

An attribute can also consist of other component attributes, in which case it is called **compound**. The *parent* of an attribute is the entity type, the relationship type or the compound attribute to which it is directly attached. An attribute whose parent is an entity type or a rel-type is said to be at level 1. The components of a level-i attribute are said to be at level i+1.

If the value domain has some specific characteristics, it can be defined explicitly as a **user-defined** domain, and can be associated with several attributes of the project. A user-defined domain is atomic or compound.

Each attribute is characterized by its **cardinality** [*i* - *j*], a constraint stating that each parent has from *i* to *j* values of this attribute. Generally *i* is 0 or 1, while *j* is from 1 to N (= *infinity*). However, any pair of integers can be used, provided that $i \leq j$, $i \geq 0$ and $j > 0$. The default cardinality is [1 - 1], and is not represented graphically.

An attribute with cardinality [*i* - *j*] is called:

- single-valued* if $j = 1$
- multivalued* if $j > 1$
- optional* if $i = 0$
- mandatory* if $i > 0$.

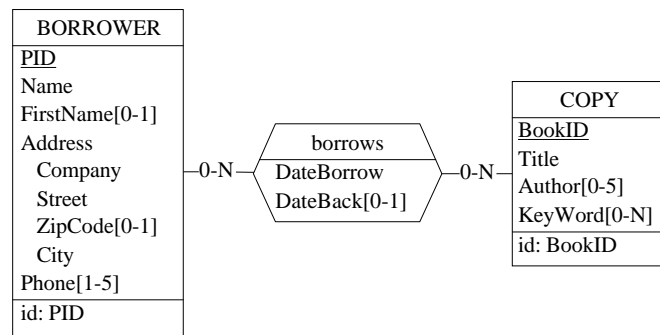


Figure A.9 - Examples of attributes. Name is mandatory [1-1] while FirstName is optional [0-1]. Address is compound while Name and ZipCode are atomic. Phone, Author and KeyWord are multivalued. The cardinality of KeyWord is unlimited [0-N].

A.11 Object-attribute

Any entity type (or object class) can be used as a valid domain for attributes. Such attributes will be called **object-attributes**. They mainly appear in object-oriented schemas.

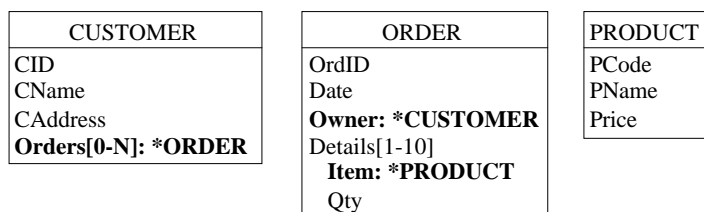


Figure A.10 - Owner is a single-valued object-attribute. For each ORDER entity, the value of Owner is a CUSTOMER entity. Orders is a multivalued object-attribute of CUSTOMER. This construct can be used in OO database schemas to express relationship types.

A.12 Non-set multivalued attribute

A plain multivalued attribute represents sets of values, i.e., unstructured collections of distinct values. In fact, there exist six categories of collections of values.

- **Set**: unstructured collection of distinct elements (default).
A[i - j] represents a collection of i to j unique values.
- **Bag**: unstructured collection of (not necessarily distinct) elements.
A[i - j] bag represents a collection of i to j elements, some of which may be identical.
- **Unique list**: sequenced collection of distinct elements.
A[i - j] ulist represents a sequence of i to j unique elements.
- **List**: sequenced collection of (not necessarily distinct) elements.
A[i - j] list represents a sequence of i to j elements, some of which may be identical.
- **Unique array**: indexed collection of cells that can each contain an element. The elements are distinct. Some cells can be empty.
A[i - j] uarray represents a collection of j cells, of which i to j can contain a value. These values are unique.
- **Array**: indexed collection of cells that can each contain an element. Some cells can be empty.
A[i - j] array represents a collection of j cells, of which i to j can contain a value. Some of these values may be identical.

These categories can be classified according to two dimensions: uniqueness and structure.

	Unstructured	Sequenced	Array
Unique	(set)	ulist	uarray
Not unique	bag	list	array

STUDENT
<u>RegNbr</u>
Name
Phone[0-2]
Expenses[0-100] bag
ChristName[0-4] ulist
MonthlyScore[0-12] array
id: RegNbr

Figure A.11 - Some non-set multivalued attributes. While Phone defines a pure set of 0 to 2 values, Expenses represents a bag of 0 to 100 values, Christ(ian-)Name a list of 0 to 4 ordered distinct values and MonthlyScore an array of 12 cells, of which from 0 to 12 can be filled.

A.13 Group

A *group* is made up of *components*, which are attributes, roles and/or other groups. A group represents a construct attached to a parent object, i.e., to an entity type, a rel-type or a multivalued compound attribute. It is used to represent concepts such as identifiers, foreign keys, indexes, sets of exclusive or coexistent attributes.

It can be assigned one or several **functions** among the following:

primary identifier: the components of the group make up the *main identifier* of the parent object; it appears with symbol **id**; if it comprises attributes only, the later are underlined in the graphical view; a parent object can have at most one primary id; all its components are mandatory.

secondary identifier: the components of the group make up a *secondary identifier* of the parent object; it appears with symbol **id'**; a parent object can have any number of secondary id. Some components can be optional.

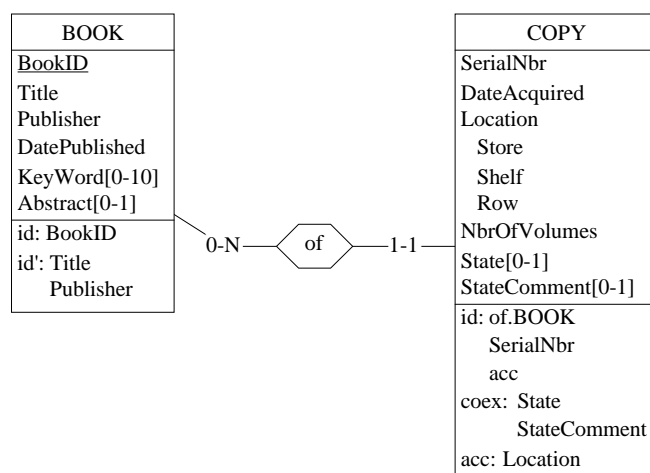


Figure A.12 - Some constraints. BookID is a primary identifier and {Title, Publisher} a secondary identifier of BOOK. SerialNbr identifies each COPY within a definite BOOK. In addition, this identifier is an access key. Optional attributes State and StateComment both are valued or void (coexistence).

coexistence: the components of the group must be *simultaneously present or absent* for any instance of the parent object; the group appears with symbol **coex**; all its components are optional.

exclusive: among the components of the group *at most one must be present* for any instance of the parent object; the group appears with symbol **excl**; all its components are optional.

at-least-1: among the components of the group, *at least one must be present* for any instance of the parent object; the group appears with symbol **at-1st-1**; all its components are optional.

exactly-1: among the components of the group, *one and only one must be present* for any instance of the parent object (= *exclusive* + *at-least-1*); the group appears with symbol **exact-1**; all its components are optional.

access key: the components of the group form an *access mechanism* to the instances of the parent object (generally an entity type, to be interpreted as

a table, a record type or a segment type); the access key is an abstraction of such constructs as indexes, hash organization, B-trees, access paths, and the like; it appears with symbol **acc** or **access key**.

user-defined constraint: any function that does not appear in this list can be defined by the user by giving it a name; some examples: **at-most-2** (no more than two components can be valued), **lhs-fd** (left-hand-side of a functional dependency), **less-than** (the value of the first component must be less than that of the second one), etc.

An identifier can be made up of a multivalued attribute, in which case it is called a **multivalued identifier**. In this case, no two parent instances can share the same value of this attribute.

A multivalued, compound, attribute A, with parent P (entity type, relationship type or compound attribute) can be given identifiers as well. Such an **attribute identifier I**, made of a subset of the subattributes of A, states that, for each instance of P, no two instances of A can share the same value of I.

CUSTOMER	ORDER	PRODUCT
<u>CID</u>	<u>OrdID</u>	<u>PCode</u>
CName	Date	PName
CAddress	Owner: *CUSTOMER	Price
Orders[0-N]: *ORDER	Details[1-10]	id: PCode
id: CID	Item: *PRODUCT	
id':Orders[*]	Qty	
	id: OrdID	
	id(Details):	
	Item	

Figure A.13 - Multivalued identifiers and Attribute identifiers. Object-attribute Orders is declared an identifier, stating that any two CUSTOMER entities must have distinct Orders values (an order is issued by one customer only). All the Details values of each ORDER entity have distinct Item values (a product cannot be referenced more than once in an order)

An identifier of entity type E is made up of either:

- *one or several single-valued attributes of E,*
- *one multivalued attribute of E,*
- *two or more remote roles of E,*
- *one or more remote roles of E + one or more single-valued attributes of E.*

An identifier of relationship type R is made up of either:

- *one or several attributes of R,*
- *two or more roles of R,*
- *one or more roles of R + one or more attributes of R.*

An identifier of attribute A is made up of:

- *one or several single-valued subattributes of A.*

A technical identifier (technical id) of entity type E is a semantic-less, generally short, attribute that is used to denote entities without reference to application domain properties. It is generally used as a substitute for long, complex and information-bearing identifiers. Object-id (oid) of OO models can be considered as system-controlled technical identifiers. The default type and length of technical ids are user-defined.

A.14 Inter-group constraint

Independently of their function(s), two groups with compatible components can be related through a relation that expresses an inter-group **integrity constraint**.

The following built-in constraints are available:

reference: the first group is a foreign key and the second group is the referenced (primary or secondary) identifier; the foreign key appears with symbol **ref**;

refequal: the first group is a foreign key and the second group is the referenced (primary or secondary) identifier; in addition, an inclusion constraint is defined from the second group to the first one; both constraints form an *equality* constraint; the foreign key appears with symbol **equ**;

inclusion: each instance of the first group must be an instance of the second group; since the second group need not be an identifier, the inclusion constraint is a generalization of the referential constraint (to be implemented);

incl equal: an inclusion constraint in each direction: each instance of each group is an instance of the other group (to be implemented);

copy: (to be defined)

copy equal: (to be defined)

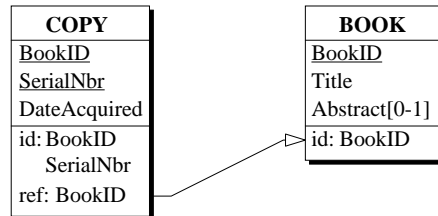


Figure A.14 - Attribute BookID of COPY form a reference group (foreign key) to BOOK

An *inverse* constraint can be asserted between two object-attributes, expressing that each is the inverse of the other.

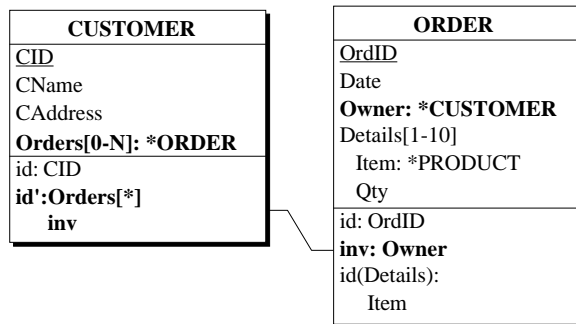


Figure A.15 - Orders of CUSTOMER and Owner of ORDER are declared inverse object-attributes. If c denotes the Owner of ORDER entity o, then c must belong to the Orders value set of CUSTOMER c.

A *generic inter-group constraint* can be drawn from any group to any other group of the schema.

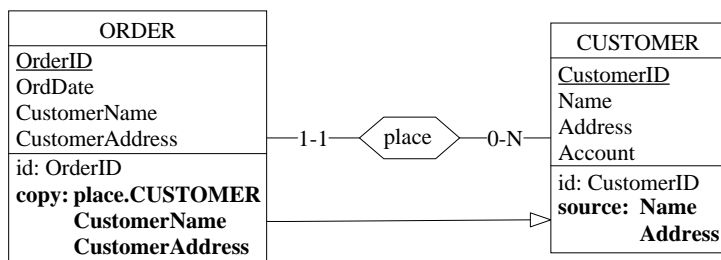


Figure A.16 - A redundancy constraint is expressed between two user-defined group types, namely copy and source, through a generic inter-group constraint. This structure states that CustomerName and CustomerAddress are copies of Name and Address of CUSTOMER through rel-type place.

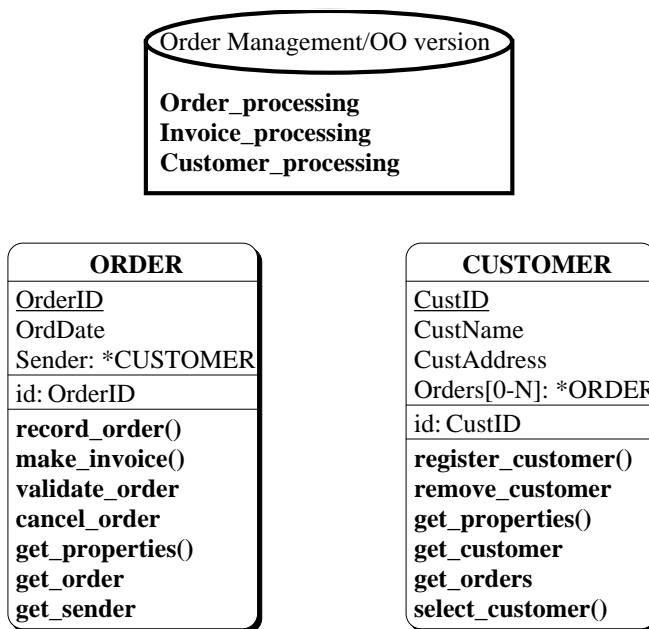


Figure A.17 - This schema includes two object classes with their methods. In addition, three global processes have been defined at the database level (attached to the schema).

A.15 Processing units

A *processing unit* is any dynamic or logical component of the described system that can be associated with a schema, an entity type or a relationship type (Figure A.17). For instance, a *process*, a *stored procedure*, a *program*, a *trigger*, a *business rule* or a *method* can each be represented by a processing unit.

A.16 Common characteristics

Some characteristics are common to several objects. Schemas, text files, entity types, rel-types, attributes, user-defined domains, collections, groups and processing units each have a *Name*, and can have a *Short-name*, a *Semantic description* (SEM), and a *Technical description* (TECH). They can also be given *semi-formal* and *dynamic properties*. Finally, they can be *marked* and *colored*.

Semantic description. The semantic description is a free text annotation explaining the meaning of the object.

Technical description. The technical description is a text giving information on the technical aspects of the object. Some functions of the CASE tool write statements in this description, especially in reverse engineering processes.

Semi-formal properties. The semantic and technical descriptions can include semi-formal properties. Such a property is declared through the statement

```
#<property-name> = <property-value>
```

where <property-name> is the name of the property and <property-value> its value. Semi-formal properties are user-defined and are not managed by the tool, but can be used by specific processors developed in Voyager-2. Defining a dynamic property is a more formal, but less flexible, way to augment the modeling power of the tool.

Dynamic properties. In addition to the built-in *static properties*, such as name, short-name, cardinality, type and length, that appear in the property box of the objects, each object type can be dynamically given additional properties, called *dynamic properties*. They are defined by the analyst at the meta-object level (schema, entity type, rel-type, attribute, etc.), in

such a way that they can be given a value for each instance of the meta-object (each schema, each entity type, each rel-type, each attribute, etc.). For instance, attributes can be associated with such dynamic properties as owners, synonyms, definition, French name, password, physical format, screen layout, etc. DB-MAIN itself maintains some internal dynamic properties. They are visible but have a read-only status.

A dynamic property has a name (`Name`), a type (`Type`), and a textual description (`Sem`). It can be updatable by the users or not (`Updatable`). It can be single-valued or multivalued (`Multivalued`). It is possible to declare the list of possible values (`Predefined values`).

Marking. Each product and each process in a project, each object in a schema and each line in a text file can be given a special status, called *marked*. Marking is a way to permanently select objects, either to identify them (e.g., validated objects are marked, while those still to be examined are unmarked), or to apply global operations on them through the assistant (e.g., transform all *marked rel-types* into entity types or export specifications) or as the result of the execution of some assistants or to define schema views.

In fact, there are several *marking planes*, numbered 1 to 5, of which one is the current, or visible, plane. A plane is a set of simultaneous marks associated with the objects of a schema. All the operations are applied in the current plane. The concept of plane makes it possible to define up to 5 independent sets of marks on the same schema, e.g., one to denote validated objects, one for import/export and one for temporary operations. It is possible to combine the marks of several planes.

Color. Selected objects of a schema can be drawn in a definite color. Several colors can be used in the same schema.

A.17 Names

The model includes naming constraints that make it possible to denote objects through their name. Here are the main rules:

- two names composed of the same characters, be they in uppercase or in lowercase, are considered identical; so, "Customer" and "CUSTOMER" are the same names; the accents are taken into account;
- all the printable characters, including spaces, /, [, {, (, punctuation sym-

bolds and diacritic characters, can be used to form names; however the symbols " is prohibited;

- the *schemas* of a project are identified by the combination <name>/<version>;
- each *entity type* of a schema is identified by its name;
- each *rel-type* of a schema is identified by its name;
- a *collection* of a schema is identified by its name;
- each direct *attribute* of a definite parent (an entity type, a rel-type or a compound attribute) is identified by its name;
- a *group* of a definite parent (idem) is identified by its name.
- each *processing unit* of a definite parent (an entity type, a rel-type or a schema) is identified by its name;

The syntax of names includes the special symbol "|", which is a valid character, but which has a special effect when displayed: this character as well as all the characters that follow are not displayed.

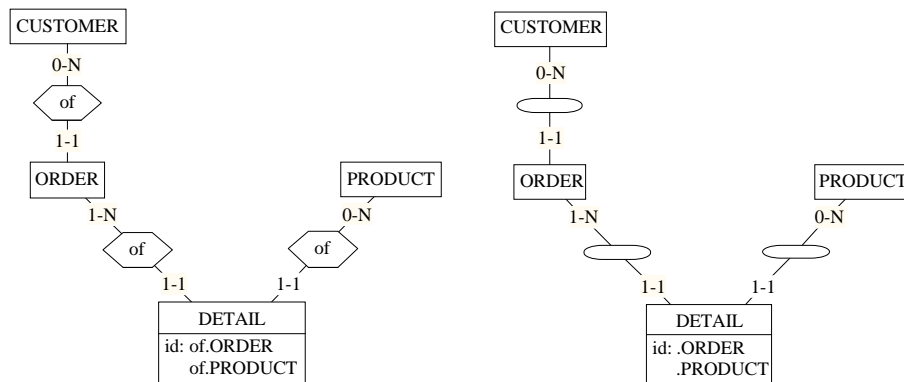


Figure A.18 - Use of ambiguous names. The rel-types have been assigned the names "of|1", "of|2", "of|3" in the left-side schema and "|1", "|2", "|3" in the right-side schema

A.18 Structure of a text file

At the lowest level of understanding, a text file is a system object containing a *string of characters*. Most files comprise *text lines*, that are logical units of text. One or several (not necessarily contiguous) lines can be *marked* in each of the five marking planes, in order to maintain up to five permanent sets of lines. An *annotation* can be associated with each line. In some circumstances, words and lines can be colored.

Texts which have a meaningful structure, such as any kind of programs, often include patterns. A *text pattern* is a formally defined text structure that can appear in several places in the text, and that is defined by a set of syntactic rules. Any section of text that satisfies these rules is a *instance* of this pattern.

Useful structures can be extracted from program files, such dependency diagrams and program slices. They will be studied in other lessons devoted to *program understanding* and *reverse engineering*.