MASTER IN COMPUTER SCIENCE

Data analysis for reverse engineering of relational databases

Lamouchi, Olfa

Award date:
2000

25
40

Facultés Universitaires Notre-Dame de la Paix
University of Namur, Belgium
Computer Science Department

Data Analysis for Reverse Engineering of Relational Databases

Olfa Lamouchi

Director : Prof. Jean-Luc Hainaut

Academic year 1999-2000

45 : def pas claire

Master thesis presented in order to obtain the title of "Maître en Informatique"

Abstract

Among the panoply of techniques and information sources available during a database reverse engineering process, this work puts emphasis on the use of data analysis to recover implicit constructs and constraints encountered in relational databases. Data analysis is a costly technique which involves complex issues. Any attempt to reduce the complexity of the various tackled problems is worthwhile. Heuristics to reduce this complexity will be proposed. A strategy to organise the reasoning is eventually presented. Then other reverse engineering techniques will be rapidly investigated and, for some implicit constructs and constraints, strategies will be proposed. Last but not least, a prototype designed to implement some of the heuristics proposed in the theoretical part will be depicted.

Résumé

Parmi la panoplie de techniques et de sources d'informations qui sont à notre disposition lors d'un processus de rétro-conception de bases de données, ce travail met l'accent sur l'analyse de données afin de recouvrer des contraintes et autres constructions implicites cachées dans des bases de données relationnelles. L'analyse de données est une technique onéreuse qui amène son lot de problèmes complexes. Toute tentative de réduire cette complexité vaut la peine d'être entreprise. Différentes heuristiques vont être proposées à cette fin. Une stratégie pour organiser ce raisonnement sera finalement présentée. D'autres techniques de rétro-conception seront ensuite examinées afin d'approfondir la recherche de certaines constructions et contraintes implicites, et de nouvelles stratégies seront proposées. Finalement, un prototype implémentant certaines des heuristiques sera décrit.

To Adhir, la rose de la famille.

Table of contents

Acknowledgements

It is a great pleasure for me to acknowledge the assistance of many people.

First, I would like to thank Professor Hainaut for his assistance, comments and for the efforts he made to push me to be more realistic. The task was arduous. It was really a learning experience for me.

I would like to thank, the DB-Main team and especially Jean Henrard for reviewing chapter 6 of this work and for his suggestions and comments and Didier Roland for his reviewing and support.

I would like to thank Doctor Shao and all the researchers at the university of Cardiff.

I would like to thank my friends Roberto Giglioli, Montassart, Moussa and Nisf-Rab for all the support and encouragement.

There are also a lot of people that I have not cited above but who helped me in various ways. I thank all of them too.

Last but not least I gratefully acknowledge the continuous support, patience, encouragement of Alain, my husband.

Chapter 1

Introduction

Reverse engineering databases is a very complex and expensive task. Every single source of information is worth to catch attention. One of them is the data handled by the database. In this work, we will survey this particular source and examine it from all possible points of view.

This chapter includes three sections: a presentation of reverse engineering, the state of the art and the objectives of this work.

Some concepts related to Database reverse engineering and the different phases involved in this process will be presented.

A presentation of the state of the art will be done, especially these works related to the reverse engineering of relational databases.

1.1. Reverse engineering

We will start this section by a definition of reverse engineering followed by the objectives of such a process.

Reverse engineering can be defined as analyzing a system to identify its components and their relationships and to create representation of the system at higher levels of abstraction. Alternatively: "reverse engineering is a process to support the analysis and understanding of data and processing in existing computerized systems" [Hainaut98].

Note that reverse engineering is just a step among others in the information system lifecycle; it is not an end in itself, its results are intended to give an aid for comprehension and to help for the maintenance. These results are also used to redocument, convert, restructure, or extend the subject information system.

The objectives of database reverse engineering (DBRE) are numerous, below we will try to give a list of the most frequent ones:

- Knowledge acquisition in system development:

  When developing a new system, one starts by gathering and formalizing user's requirements. It is frequent to have some partial implementation of the future system. The analysis of this system could bring useful information.

- System maintenance:

  To be able to find out the bugs and to modify the system functions, one needs a good understanding of the concerned component. The benefits brought by reverse engineering are here primarily in terms of maintenance cost savings. There has been recognition that

maintenance is not just concerned with fixing the bugs. Indeed, there exists a range of maintenance types:

Corrective maintenance involves the correction of the errors discovered during the system life.

Adaptative maintenance is concerned with the adaptation of the system to environmental changes, e.g. new hardware, system software, laws,...

Perfective maintenance involves the improvement of the system according to some new requirements and its enhancement by adding some functionalities.

Preventive maintenance is concerned with the updating of the system to prevent future problems.

Each of these different types of maintenance involves interrogating the subject system and documentation.

- System reengineering:

  It is the process of reverse engineering a system to a certain level of abstraction and then reconstructing it. It consists on changing the internal architecture of the system or rewriting the code of some components without modifying the external specifications.

- System extension:

  It is concerned with changing and enhancing the functionalities of a system.

- System migration:

  It consists in replacing one or several implementation technologies, for example, Cobol / C, centralized / distributed,...

- System integration:

  The cooperation between several information systems may arise in a number of cases. Different organizations may want to cooperate. Two companies may merge, or an organization may have several information systems developed independently and new managerial needs push them to interoperate.

  Integration of two or more systems is to consolidate a new one that comprises all of them in a consistent, coherent and complete way.

  The new system could be a physical or a virtual system (e.g. federated database architecture), in this case a dynamic interface is used to translate the queries into local queries to be applied on the source systems.

  Reverse engineering could help in this case and it facilitates the integration.

- Quality assessment:

  Hints about the quality of a system could be derived from an analysis of its code and its data structures.

  According to Blaha " Reverse engineering provides an unusual source of insight. The quality of the database design is an indicator of the quality of the software as a whole " [Blaha97].

- Data extraction/conversion:

Knowledge about the physical and semantic characteristics of the data to be converted into another format is needed when the only source of information available is the database.

- Data administration:

  To develop a data administration function, we need to know and record the description of all the information resources of the organization.

- Component reuse:

  In emerging system architectures, reverse engineering allows developers to identify, extract and wrap legacy functionalities and data components in order to integrate them in new systems.

- Documentation:

  Reverse engineering can elucidate poorly documented and undocumented existing softwares when the developers are no longer available for advice.

  It has the same role when the documentation is available but is out of date.

  As already mentioned, the documentation produced by the reverse engineering process can greatly assist maintenance.

1.2. Specific DBRE problems[1]

Database reverse engineering is to yield the complete logical schema and the conceptual schema of the database.

This process is more complex than just analyzing Data Description Language (DDL is the part of database management system facilities by which the declaration and the built of the data structures of a database is made). The information included in the DDL code is not sufficient to capture the whole structures and semantics of the system, other information sources have to be explored to get the structures and constraints that have been untranslated. Besides that, one could face non standard implementation techniques and bad designed schemas that add another degree of difficulty.

The task is particularly arduous when one deals with old and poorly documented applications.

The most frequent sources of problems that have been identified are:

- Weakness of the DBMS models:

  The technical model provided by the data management system can express just a subset of the structures and constraints expressed in the conceptual schema.

  The discarded constructs might be managed in procedural components of the application such as programs, triggers...

- Implicit structures:

  It seems to be a choice of the designer not to declare these structures in the DDL. They are generally implemented by the mean of procedural components.

[1] Derived from [Hainaut98].

- Optimised structures:

  For technical reasons, many database structures include non-semantic constructs; redundant and unnormalized constructs could be added to improve response time.

- Awkward design:

  Novice and untrained developers often produce poor or wrong structures.

- Obsolete constructs:

  The current programs can ignore some parts of a database.

- Cross model Influence:

  The professional background of designers can lead to very strange results. For instance some relational databases are translations of Cobol files or of spreadsheets.

1.3. State of the art

Reverse engineering is of increasing importance these days and several researches have been done in this field. The complexity of the process of reverse engineering gives raise to various approaches.

Some of them aim at recovering a schema in a conceptual model which is, most of the time, either the Entity-Relationship model (ER), in one of its variants such as the Extended Entity Relationship model[2] (EER), or an Object Oriented (OO) model.

Others have worked on special issues without trying to get up to the conceptual model. In this category we find works on functional dependencies, inclusion dependencies and multivalued dependencies.

Note that Reverse engineering transformations are not as straightforward as forward engineering ones, this is because each source schema can be transformed into several different schemas.

We are interested here by the works concerned with relational databases.

Up to now, a generally accepted methodology for reverse engineering has not been established, but we can classify the approaches concerned with the translation of relational databases to conceptual models into two sub-classes:

In the first sub-class, the approaches elicit the semantics of a given relational schema by evaluating its *inclusion dependencies*[3]. Each inclusion dependency is interpreted according to the role of its attributes (key, part of key, foreign key, non key). Most of these approaches require the complete set of inclusion dependencies and keys of the relations, some take into account only those inclusion dependencies that are key-based [Fahrner 95].

The approaches in the second sub-class derive an ER schema through an evaluation of *keys*, their construction through other keys, and a discovery of foreign keys. Notice that the name semantics is of prior importance here, since relationships between keys are mainly identified through their names. These approaches are applicable only if keys and attribute names are given. The evaluation of the name semantics can be considered as a heuristic to derive foreign keys.[Fahrner95]. Several of these approaches also classify the relations with respect to the

---

[2] For more detail about the EER model, see [Chiang 94].

[3] See chapter5.

construction of their keys. Most of the approaches of this category assume relational schemas are in third normal form[4].

The approach of Chiang differs from the others, since the only prerequisite is the existence of a database instance. It is a mixture of a key-based and of an inclusion dependency-based approach, indeed, the relations are classified first w.r.t their primary keys and then inclusion dependencies are generated by applying heuristics.

We will try below to explore in some details some of the works done in the field:

[Navathe 87]

The authors propose two procedures, one maps a database schema from the relational model into an Entity Category Relationship model[5] (ECR) and the other one is a mapping from the hierarchical model into the ECR model. We are interested here only by the mapping from the relational to the ECR model. The relational schema has to be in 3 NF (if it is not the case, it is normalised first) and possibly 2 NF.

They start by renaming the primary and candidate keys (the user is involved in this step). The input relations and their attributes are classified, then the mapping process takes place. The cardinality of binary relationships is derived by checking the dependencies between the keys of the entity types that are linked by this relationship.

When some entities have the same attributes as their primary key, the user is asked whether there are any subclass or superclass among such entities.

[Davis 88]

This work presents an algorithm that converts the relational model to the ER model. It assumes that all relations are in 3NF. Only the primary keys are considered.

Tables with single or composite keys are translated into entity types. Tables with dangling keys (where a dangling key attribute is an attribute that is included in the primary key of the relation but not in that of the correspondent entity) are converted to weak entity types. She considers also the foreign keys, to get the relationship types. She does not specify the information sources.

[Markowitz 90]

In this work a procedure for translating relational schemas into Extended Entity Relationship (EER) schemas is developed. It first transforms a relational schema into an appropriate form for identifying EER objects structures, thus certain cyclic inclusion dependencies are detected and removed, then every relation is mapped into an object-set (e.g. weak entity, specialisation entity...). By examining the set of inclusion dependencies (INDs) and the structure of the keys in each relation, he derives the type of the object-set and the type of the object-set connections. The INDs considered here are key-based ones.

[Johannesson 94]

Johannesson presents a method for translating a relational schema into the EER conceptual schemas. He assumes that all relations are in 3NF. He starts by transforming the relational schema into an appropriate form for identifying object structures. He considers primary keys,

---

[4] For more details about normal forms, see chapter five.

[5] For more details about the ECR model, see [Navathe 87]

candidate keys, foreign keys and inclusion dependencies where both sides are keys. This later gives rise to either mandatory attributes or generalisation constraints; this choice is done by a user having knowledge about the semantics of the relational model.

The object structures are identified on the basis of the interactions between keys and inclusion dependencies.

The only explicit source of information is the participation of the user.

[Permalani 94]

In this article, the authors adopt the Object Modelling Technique[6] (OMT) notation for modelling data. There are three sources of information: schemas, data analysis and semantic understanding of application intent.

They use the candidate keys for the analysis rather than primary keys. They accept schemas in a normal form less than 3NF. The candidate keys are derived from unique indexes, automated scanning of the data and the semantic knowledge. If the foreign keys are not specified in the DBMS, foreign key groups (a foreign key group is a group of attributes within which FKs could be found) can be deduced (after resolving homonyms and synonyms) by investigating matching names, data types and domains; information about FKs could be found in view definitions and secondary indexes. They also deduce the cardinalities of the associations. They use data analysis to confirm the hypotheses about the discovered generalisations.

[Kalman 91]

An algorithm that converts a relational database to an ECR model is presented. The input of the program is a relational schema in the 3NF. In this work, the user interaction is needed.

Relations are classified, with the help of the user, according to their primary keys. On the basis of this classification, the relational schemas are interpreted and transformed. The relationships are derived from the analysis of keys and their inclusions. He considers candidate keys in a pre-processing step by replacing the primary key by a candidate key in some special cases, for example, in the case that a candidate key of a relation is found to be equal to a primary key of another relation and if there exists a subclass-superclass relationship the two relations.

[Tari98]

The authors address the problem of recovering OO schemata from relational databases. Their methodology involves two main steps:

A classification of relations to reflect the different OO constructs and consists in partitioning relations into three categories: base relations which form the core classes of the target OO schema i.e. relations that do not contain FKs, dependant relations that describe binary relations between classes and, finally, composite relations that describe ternary relationships between classes.

The identification of the different types of relationships between classes is based on:

- the analysis of relation keys (from the relational schema).
- the analysis of constraints defined explicitly in referential integrity constraints and implicitly in data sources.
- The translation

---

[6] For more details about the OMT notation, see [Permalani 94]

The relational schema is assumed to be in 3NF.

They use the relational schema and data analysis. At the schema level, the correlation between relation keys is analysed. At the data level, the correlation between data store is checked.

[Andersson 94]

It is an approach based on the use of data manipulation statements to extract the semantic information stored in a relational database.

By the analysis of equi-joins clauses, he identifies the semantically related attributes which represent references between the relation schemas in the database. The resulting information is represented in a connection diagram. No assumption is made about the normal form, no information is needed for the keys (i.e. he will not use the data dictionary to determine the keys).

Cyclic joins can be used to draw conclusions about keys e.g. attributes that are tested against themselves for equality cannot be keys and if the query does not include the keyword *distinct* and if no cursor or similar construction is defined on the result that means the attributes in the selection criteria are primary keys.

The data manipulation statements can also be analysed w.r.t *where-in* clauses (the equivalence for the join condition in SQL).

The use of the *distinct* keyword in a query implies that the attributes have non unique values and the attributes on which a *group by* clause are defined are not keys.

The data extension is examined to establish a functional dependency graph and an inclusion dependency graph.

[Chiang 94]

The authors present a methodology for translating relational database to the Extended Entity Relationship (EER) model.

The assumption about the input database are:

- At least 3NF.
- Consistent naming of key attributes.
- No error data in values of key attributes.

The user, data schema and data instances are the sources of information. This methodology is made of four steps:

1. Decomposition of input relations into at least 3 NF.
2. Classification of relation and attributes.
3. Generation of inclusion dependencies.
4. Identification of modelling structures of the EER model.

Primary keys are obtained by querying the target DBMS or are specified by the user. Candidate keys are considered only when there are ambiguities in the classification of relations and/or when the user specifies inclusion dependencies (INDs) between non key attributes. Thus, the data instances are analysed to verify the proposed INDs and to identify the candidate keys.

Note that some heuristics are used to:

- Propose possible primary key attributes for each relation.
- Formulate possible key-based INDs.
- Specify default cardinality ratios for binary relationship types.

[Hainaut 97]

A generic methodology for database reverse engineering is proposed [Hainaut 97]. By *generic*, we have to understand that it could be applied to various file management systems and database management systems.

The main processes of this approach are:

- Data structure extraction: Which consists in recovering the complete DMS (Data Management System) schema, including all the implicit and explicit structures and constraints. A first-cut logical schema will be provided, which has to be refined by further analysis of some other components of the system such as screen and reports, views, subschemas, procedures, program execution...
- Data structure conceptualisation: This phase deals with the conceptual interpretation of the DMS schema. It consists, among others, in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimisation and DMS-dependent constructs. It consists of two sub-processes, namely Basic conceptualisation and conceptual normalisation.

This approach will be explained in more details in further chapters.

We will end this section with a summary of most of the different approaches explained above.(see Table 1). Note that the symbol **?** stands for no information is available.

Approach	The target Conceptual model	Basis of the methodMethod	Input	Involvement of the user
[Navathe 87]	ECR	Keys	Keys, 3NF	Yes
[Davis 88]	ER	Keys	Keys, 3NF.	No
[Markowitz 90]	EER	INDS	Keys, FD, FK.	No
[Kalman 91]	ECR	Keys	Keys,3NF, proper naming.	Yes
[Johannesson 94]	EER	INDS	Keys, INDs, 3NF.	Yes
[Permalani 94]	OMT schema	Candidate keys	Instances, semantic understanding of application intent.	?
[Tari]	OO	?	Keys,3NF,instances.	?
[Andersson 94]	ECR	?	Data manipulation statement, instances.	?
[Chiang 94]	EER	Keys and INDs	Instances.	Yes
[Hainaut 97]	ER	None	Instances, views,screen...	Yes

Table 1- Summary of the different approaches dealing with the reverse engineering of relational databases.

1.4. Objectives of this work

The aim of this work is the use of *data analysis*, in a reverse engineering project, to recover implicit constructs and constraints that could be embedded in relational databases.

Among the different techniques concerning the recovery of implicit constructs and constraints, data analysis is the one we will emphasis on, without neglecting other techniques contributions. Data analysis could be used lonely or coupled with other techniques where either the results provided by data analysis are used as input for the variety of techniques used or conversely.

This work will stress the use of data analysis not only to *prove* or *disprove* a hypothesis but also to *discover* rules when no information is available and when the main source of information is data.

Although data analysis is a very costly technique, it provides information that is impossible to gather from other sources such as what we can call *fuzzy information*. An example for that is the recovery of a foreign key, which is true to 65%.

One of our goals will be to help recovering a relational schema that will be as complete and comprehensive as possible. It will be achieved by eliciting the implicit constructs that could be found in a relational database such as functional dependencies, foreign keys…A sub-goal will be to produce results that could be considered as part of the (re)documentation of the system being reverse engineered.

We will also use data analysis to help us in discovering the alteration of constraints or structures found in the DDL; i.e. a field is declared NULL; whereas the instances show two things: first, the field has no NULL values, and, second, the presence of a special value that is intended to simulate the Null value.

This work is realised in the framework of the DB-Main project. A prototype is intended to be integrated into the DB-Main CASE tool. It will be built to measure the performances and to test the theoretical results.

To meet the different goals, we will make use of heuristics which are guided by the needs of the user and could use some prerequisites.

Chapter 2

Database reverse engineering methodology[7]

The approach adopted here considers that the complexity of the process of reverse engineering concerned with applications whose central component is a database or a set of permanent files can be reduced by considering that the files or the databases can be reverse engineered, in most cases, independently of the procedural part.

The reasons of this claim are among others:

- The semantic distance between the conceptual specifications and the physical implementation is most often narrower for data then for procedural parts.
- The permanent data structures are generally the most stable part of applications.

When the structure of the data has been recovered, it is easier to deal with the reverse engineering of the procedural part.

It is argued [Hainaut98] that dealing first with reverse engineering of the data components of the application increases the chances of success and is more efficient than dealing with the whole application. The data reverse engineering process is divided into two sub-processes namely *data structure extraction* and *data structure conceptualisation.*

Whereas the data structure extraction process is intended to recover the complete logical schema, the data conceptualisation process recovers the conceptual schema. Let us note that in this work, we will be only interested in the first process i.e. the data structure extraction process.

We can as a first step consider that the process of reverse engineering is nothing else than the reverse of forward engineering which is a series of transformations starting with the user's requirements and ending with the production of a code.

Under this hypothesis we can say that:

$$CS = FE^{-1}(code).$$

Where *CS* stands for conceptual schema, *FE* represents the function that has lead us from the conceptual schema to the code and *code* stands for the operational code.

We will not have the same analysis for systems developed according to an ideal approach and those developed according to some empirical approach. The main difference between the two

7 This chapter is derived from [Hainaut98]

systems is that for the second type a part of the semantics is not comprised into the code part but could be in the environment of the application or in the data itself.

We will introduce the database forward engineering and the different steps involved in this process. The understanding of these steps is essential to understand the steps of reverse engineering.

2.1. Database Forward Engineering

Because database design is one of the multitude of activities in the development of an information system lifecycle, we will start by a reminder about the information system lifecycle.

2.1.1. Information system life cycle

The database system is part of the information system of an organisation. This information system normally includes different kinds of resources involved in the management, use and collection of the information resources of an organisation. The information system lifecycle is often called *macro lifecycle*, whereas the database system lifecycle is called *micro lifecycle* [Elmasri00].

The distinction between the macro lifecycle and the micro lifecycle is fuzzy for information systems which major component is a database. The macro lifecycle includes six steps or activities as shown in figure1.

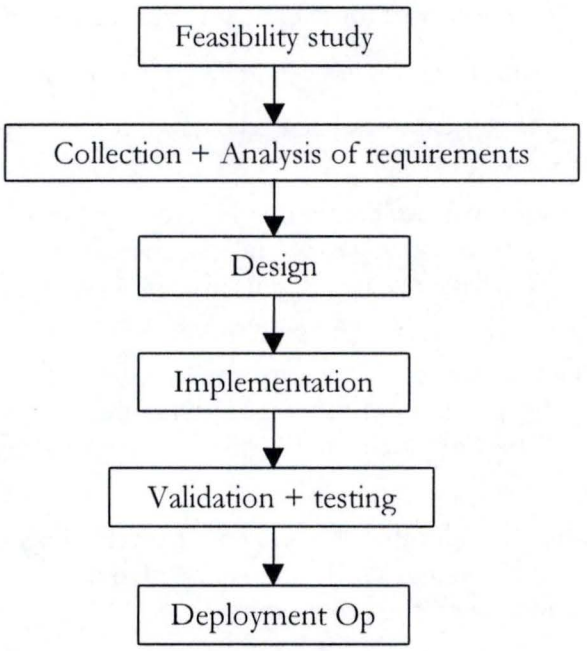


Figure 1-Information system life cycle

- Feasibility study: It is intended to define the costs of the different solutions and to set up priorities among applications.
- Requirement collection and analysis: It consists in the definition and study of the properties of the information system [Atzeni99]. The interaction with potential users is required; the output will be a complete but informal description of the data and of the operations to be carried out on it.

- Design: This phase is divided into two activities, which could be carried out simultaneously or consecutively: database design and operational design that uses and processes the database. The first activity is more concerned with the structure and the organisation of the data whereas the second one is concerned with the definition of the characteristics of the application programs.
- Implementation: It consists in the creation of the information system according to the output of the previous activity or phase; the database is constructed and loaded and the programs are coded.
- Validation and acceptance testing: The system is tested against performance criteria and behaviour specifications.
- Deployment operation and maintenance: It requires only management and maintenance operations.

2.1.2. Database Design

We will deal now only with those steps of the information system lifecycle that are close to databases. The problem of database design can be formulated as follows:

"Design the logical and physical structure of one or more databases to accommodate the information needs of the users in an organisation for a defined set of applications" [Elmasri00]. Five phases of the database design process can be identified (see figure2).

Before starting the design of the database, users requirements are analysed i.e. the expectations of the users and the intended uses of the database must be analysed. The identification of the other parts of the information system which will interact with the database system have to be carried out in order to allow the specifications of the requirements. We will explain here below some of the phases involved in database design (see figure3).

- Conceptual design or conceptual analysis

It involves two parallel activities. The first activity has as purpose to represent the data requirements produced by the analysis of users's requirements, in terms of formal and complete description independently of the DBMS to be used. After translating the semantics of the information system into conceptual structures, a normalisation on these structures will be carried out to give them enhancement in terms of quality such as normality, minimality, extendibility... The product of this phase will be a *conceptual schema* which refers to a conceptual data model such as an Entity Relationship (ER) model. The second activity [Elmasri00] will be the transaction design with the purpose of designing the characteristics of database transactions in a DBMS–independent way.

- Logical design

It consists in mapping the conceptual schema to the data model of the chosen DBMS. The product will be a *logical schema* that refers to a logical data model such as the relational data model. The data are still represented independently of the physical details. Knowing the operations to be carried out on the data, some optimisations could be done, such as minimising response time or minimising disk space... It exists a range of optimisation techniques such as discarding constructs, leaving aside identifiers and foreign-keys, splitting tables, adding structural redundancy which consists in adding new constructs in a schema such that their instances can be computed from instances of other constructs (for example attribute duplication), aggregated values representation, normalisation redundancy and restructuration...

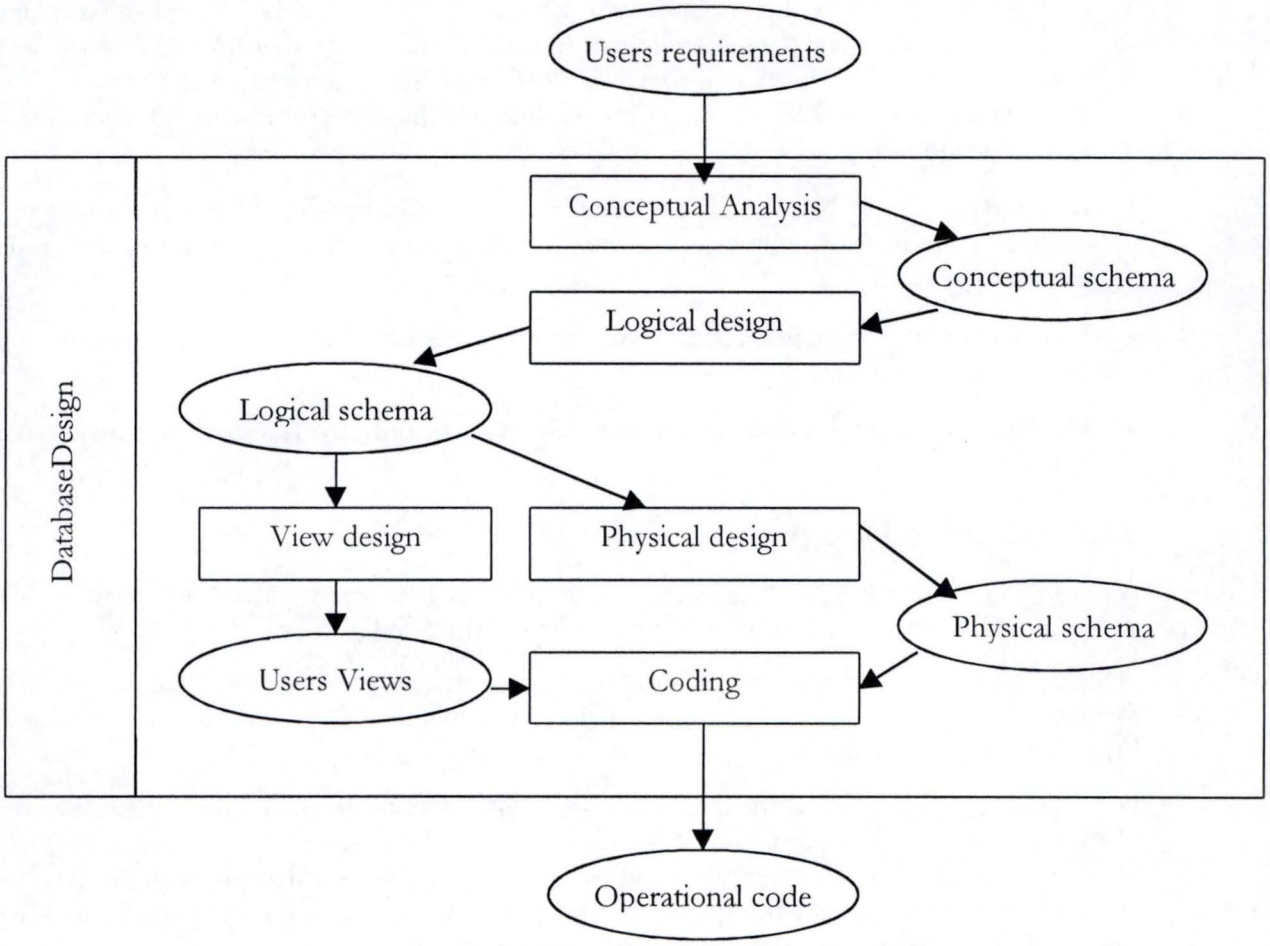


Figure 2- the main activities of database design

- Physical design

"It is the process of choosing specific storage structures and access paths for the database files to achieve good performance for the various database applications" [Elmasri00].

- Coding

Because the DMS cannot cope with all the structures and constraints, we will have two kinds of operational code, $code_{ddl}$ which comprises the explicit declaration of constructs, expressed in the DDL and $code_{ext}$ which comprises all those structures and constraints not supported by the DMS. To express and code the non-DMS constructs, we have several techniques such as predicates, views with check option, triggers, stored procedures, access module, pre-validation programs, post-validation programs…

We will try to formalise the forward engineering process and we can say that:

$$code = FE\ (CS).$$

where :

code denotes the operational code = $code_{ddl} \cup code_{ext}$.

CS denotes the conceptual schema.

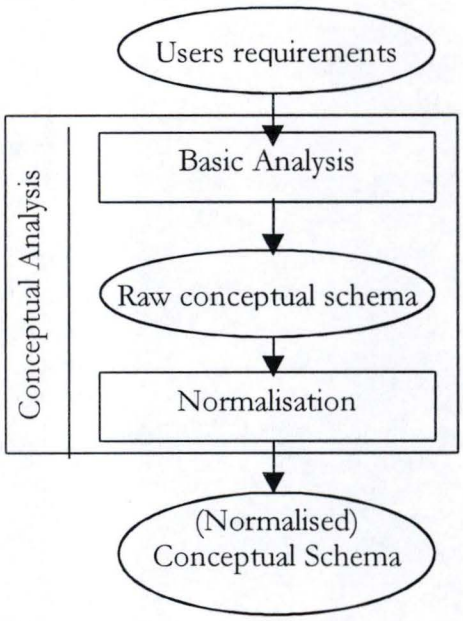


Figure 3-1 Conceptual Analysis

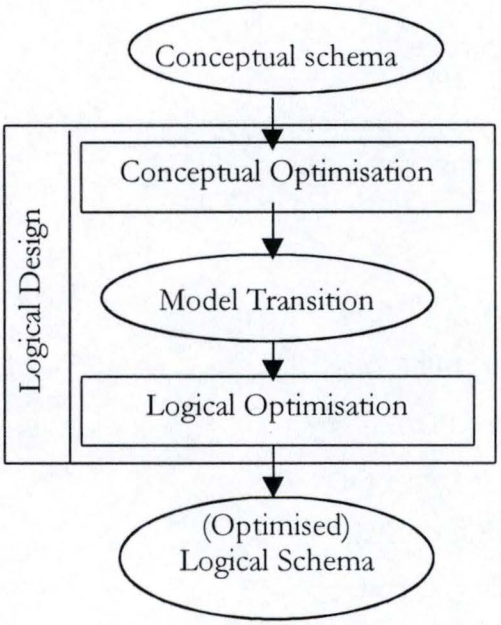


Figure 3-2-logical design

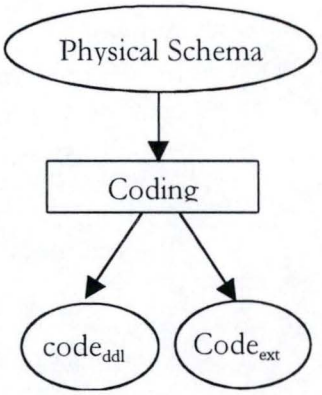


Figure 3-3-Coding

Figure 3-Conceptual Analysis, Logical Design and Coding.

Let LS and PS be consecutively logical schema and physical schema.

Let LD, PD and COD be consecutively the logical design phase, physical design phase and coding phase. We can express the transformations of the forward engineering as follows:

$LS = LD\ (CS)$

$PS = PD\ (LS)$

$Code = COD\ (PS)$.

With

$COD = \{COD_{ddl}; COD_{ext}\}$

$LD = \{OPT; TRANS\}$.

Where :

OPT: represents the processes of optimisation on the logical and conceptual levels.
TRANS: represents model translation.

Note that the notation S = S1 ; S2 means to carry out the process S implies to carry out S1 and S2 in any order.

For *an Ideal Approach* of forward engineering we will have:

$code = FE\ (CS)$

with $FE = COD \circ PD \circ LD \Rightarrow FE = \{ COD_{ddl}; COD_{ext} \} \circ PD \circ \{OPT; TRANS\}$.

Where the notation $S = S2 \circ S1$ means that an order is expected and to carry out S we have to first carry out S1 and to carry out S2 on the result obtained.

In an *Empirical Approach* things are more complicated than in ideal approaches; indeed, a part of the semantics (Δ) is outside the system ($E(\Delta)$).

Where:

$\Delta = \Delta_l \cup \Delta_p \cup \Delta_c$

Δ_l: conceptual specifications ignored during LD.

Δ_p: the semantics ignored during PD.

Δ_c : the semantics discarded in COD.

We will have:

$Code \cup E(\Delta) = FE(CS)$.

$LS \cup E(\Delta_l) = LD(CS)$.

$PS \cup E(\Delta_p) = PD(LS)$.

$Code_{ddl} \cup E(\Delta_c) = COD_{ddl}(PS)$.

$Code_{ext} = COD_{ext}(PS)$.

$Code = code_{ddl} \cup code_{ext}$.

2.2. The DBRE approach

The suggested DBRE approach consists of three phases(see figure 4):

- Extraction of PS: Its consists in recovering the physical schema from the operational code.
- Recovery of LS: It consists in recovering the logical schema from the physical one.
- Conceptualisation of LS: From the logical schema, the conceptual schema will be derived; it consists in two activities namely *de-optimisation* of LS, which deals with removing and transforming the optimisation constructs, and *unstranslation* of LS which cope with the interpretation of the logical constructs in terms of conceptual structures.

Note that the symbol ***inv*** used in ***figure4*** means that each process is the inverse of the other and that the symbol **=** means the two processes are the same.

For those systems that are developed with respect to an ideal approach we have:

$CS = FE^{-1}(code)$.

$FE^{-1} = \{OPT^{-1}; TRANS^{-1}\} \circ PD^{-1} \circ \{COD_{ddl}^{-1} ; COD_{ext}^{-1}\}$.

For the system developed according to an empirical approach we have:

$CS = RE(code \cup E)$.

RE = conceptualisation o extraction.

extraction = schema cleaning o schema refinement o DDL code-extraction.

conceptualisation = normalisation o {de-optimisation ; unstranslation } o preparation.

$LS = extraction(code \cup E(\Delta))$.

CS = conceptualisation (LS).

EPS ≡ explicit-physical-schema = DDL-code-extraction ($code_{ddl}$).

CPS ≡ complete-physical-schema = schema refinement (EPS, $code_{ext} \cup E$).

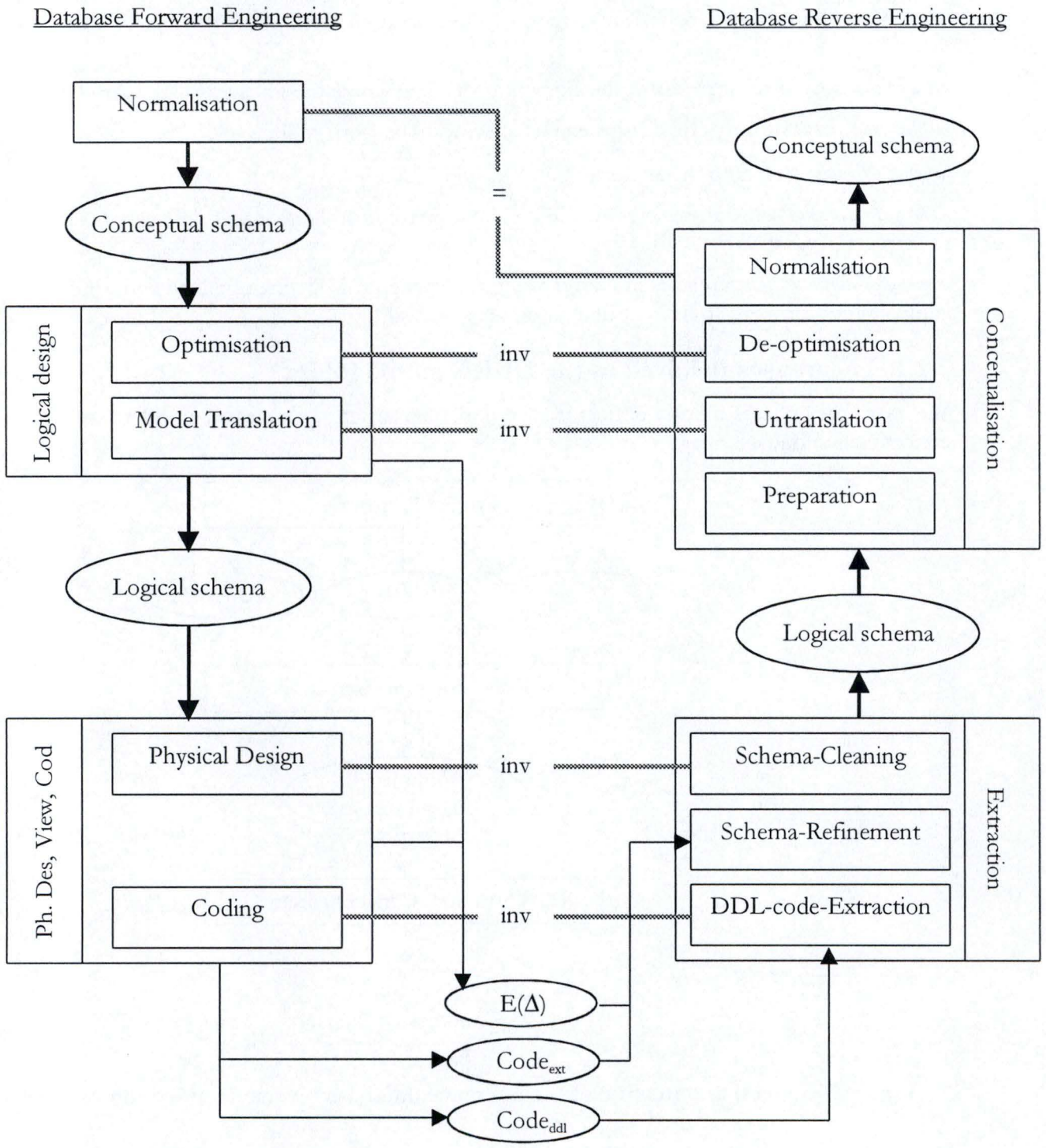


Figure 4-The main DBRE processes seen as the inverse of forward engineering process.

LS = schema cleaning (CPS)

RCS ≡ raw conceptual schema = {de-optimisation ; unstranlation } o preparation (LS).

CS = normalisation (RCS).

As indicated in the formula above: we notice that more steps are involved in the second kind of systems than those involved in systems of the first kind; Each of these steps will be presented.

schema cleaning is a sub-process of the extraction step and where the PS is translated into LS.

DDL code extraction: its purpose is to extract specifications from $code_{ddl}$

schema refinement refers to the analysis of code $_{ext}$ and E(Δ).

schema preparation is a sub-process of the conceptualisation process; it will deal with awkward constructs of LS if any.

conceptual normalisation: consists in restructuring the conceptual schema obtained from the conceptualisation process, to make it fit a corporate standard.

2.2.1. The phases involved in the DBRE methodology

We have three phases namely project preparation, data structure extraction and data structure conceptualisation.(see figure 5)

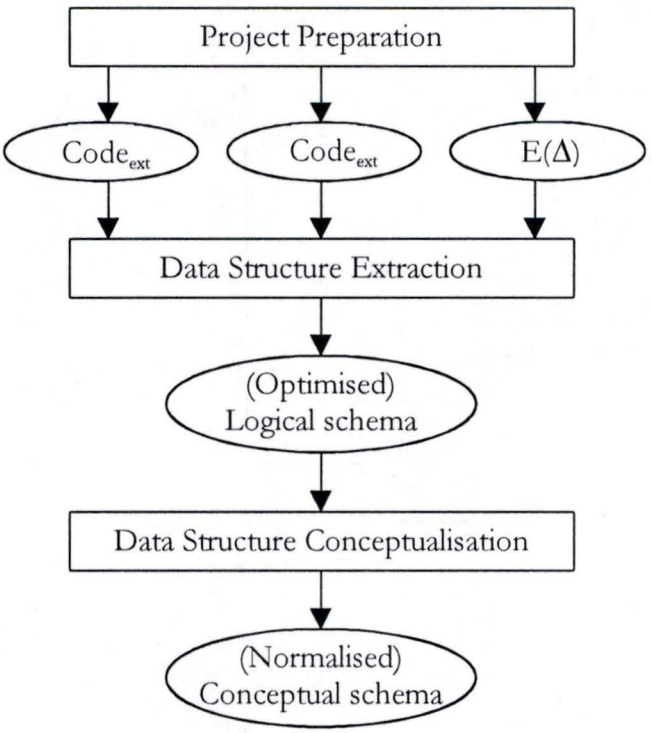


Figure 5-General architecture of the reference database reverse engineering methodology.

A. Project preparation

Its purpose is to identify and evaluate the components to analyse, to evaluate the resources needed and to define the operations planning. It is made of five sub-processes:
System identification: its purpose is to identify and to evaluate usefulness and completeness of different sources of information such as files, program sources, documentation, screens, reports, forms…

- Architecture recovery: it aims to draw both the procedural and data components of the system and their relationships.

- Identification of the relevant components: its aim is to discard the components that bring no information.
- Resource identification: it identifies the human, the technical resources and the budget…

B. Data structure extraction

Its goal is to produce a complete logical schema including all the structures and constraints. The main problem to tackle here is the discovery of the implicit constructs and to make them

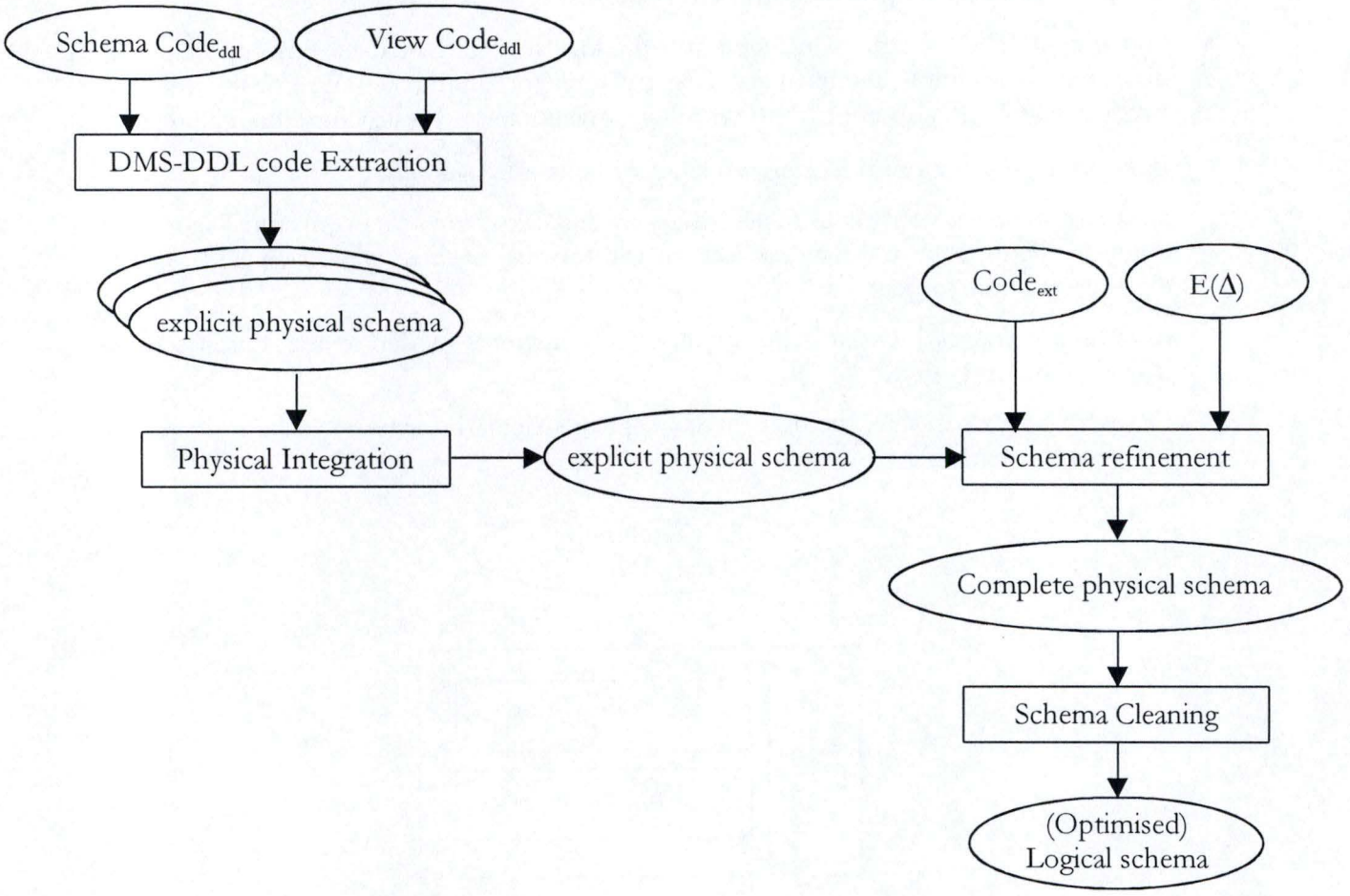


Figure 6- General architecture of the data structure extraction phase

explicit. It includes four main processes: (see figure 6)

- DMS- DDL code extraction: It produces a first-cut logical schema by analysing the data structure declaration statements from the schema scripts and application programs.
- Physical integration: It is to envisage when more than one DDL source were processed. In this case, each extraction will provide a partial schema; the final schema must include all the partial ones. This process produces the explicit physical schema.
- Schema Refinement: It produces the complete physical schema by adding to the explicit physical schema implicit and lost constructs; it is a complex process because of the variety and complexity of the information sources. The code$_{ext}$ and E (Δ) are analysed looking for hints/evidences of implicit constructs and even lost ones; by variety of information

sources we mean and think about what could be found in the $code_{ext}$, procedural sections in the application programs, forms, reports, triggers and stored procedures...

- Schema cleaning: It will produce the complete logical schema by moving away the technical constructs such as indexes and clusters.

C. Data structure conceptualisation

It will specify the semantic structures of the logical schema produced by Data structure extraction process into a conceptual schema. It is formed by three processes:

- Preparation: The constructs included into the logical schema that have no semantics are discarded, for example, the technical data structures (program counter...) or the dead data structures. It could also improve the naming conventions and restructure the schema.
- Basic conceptualisation: It tackles two kinds of tasks (see figure 7):

  schema unstranslation consists in transforming the logical schema to a conceptual schema by trying to identify the translations done in the forward engineering design and applying their reverse transformations.

  schema de-optimisation looks for the optimisation constructs hidden in the logical schema, which are de-optimised.

  Conceptual normalisation restructures the conceptual schema in order to make it expressive, simple and extendable.

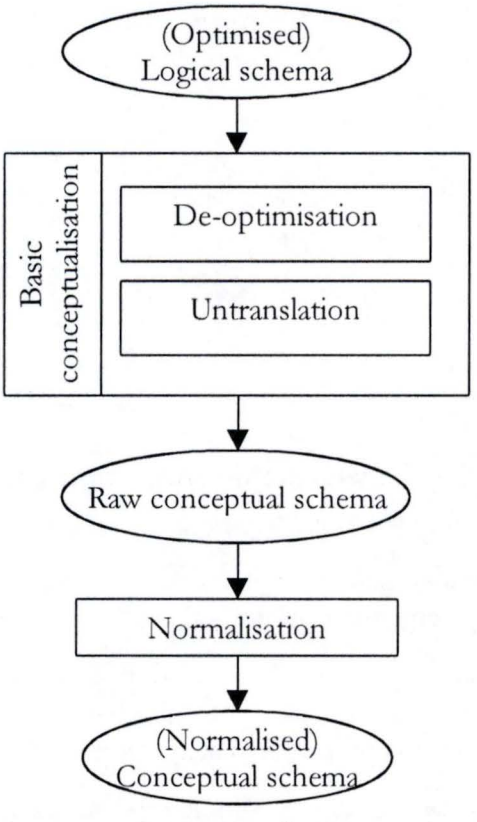


Figure 7-General architecture of the Data Structure Conceptualisation phase.

Chapter 3

Implicit constructs[8]

Our aim in this work is to recover the structures and constraints that are not specified in the Data Definition Language (DDL) files. We will give a definition and a survey of these implicit constructs and the variety of reasons and practices behind the adoption of such tricks.

3.1. Implicit constructs

3.1.1. Definition

A component or property of a data structure that is explicitly declared through a specific statement of a DDL file is referred as an explicit construct while an implicit construct is defined as:

" A component or property that holds in the data structure, but that has not been declared explicitly " [Hainaut98].

Although the Database Managing System (DBMS) participates to the management of the implicit constructs, it is unaware of their existence.

3.1.2. Example

Let Players and Team be two relation schemas defined as follows:

Players (num:integer, name:char, address:char).

Team (teamnum:integer, id:integer).

Where num and teamnum are primary keys consecutively of Players and Team. The analysis of the data shows the existence of a foreign key not declared, Team.id targeting Players.num.

3.2. Sources of implicit constructs

Five sources of implicit constructs have been identified. Their origin is the combination of the habits of the programmers and the expressiveness of the available DBMSs. Indeed, some of the constraints expressed in the conceptual schema are not supported by the DBMSs such as data dependencies.

---

[8] This section is derived from [Hainaut98].

3.2.1. Structure hiding

Structure hiding deals with structures and constraints that are supported by the DBMS but the programmer has preferred to declare them under another form that is more general and less expressive. The reasons behind that could be reusability, genericity, simplicity and efficiency.

For example, a one-to-many relationship could be implemented as a many-to-many relationship through a record type, or by an implicit foreign key.

3.2.2. Generic expression

Some DMS offer general functionalities to express and to verify constraints. As a consequence, a problem arises and resides in the fact that a constraint could be expressed in a variety of ways; this makes the task of recovering it more complex i.e. if we want to recover such a constraint we have to envisage the different kinds of implementation.

Examples of this general functionalities included in relational DBMSs are triggers, stored procedures and views with check options.

3.2.3. Non declarative structures

Non declarative structures are the structures that are not supported by the DBMS. As a consequence, they have to be represented and checked by other means that differ from the DBMS. They are often checked by procedural sections in the application programs or in the user interface, which are often not centralised.

3.2.4. Environmental properties

Sometimes, the properties of the data are guaranteed by the environment of the system and some programmers choose not to enforce these properties and/or not to translate them. So, their elicitation cannot be based on data structure extraction and program analysis.

3.2.5. Lost specifications

Lost specifications are the facts that have been moved away or ignored, intentionally or not, by the programmer.

3.3. Main implicit constructs

A large range of implicit constructs exists and could vary from a DMS to another.

Hereafter one can find the main implicit constructs that can occur in large reverse engineering projects, as well as in smaller ones.

3.3.1. Finding the fine-grained structure of attributes

An attribute declared as atomic could be a concatenation of some other attributes. It is the case when one transforms a multivalued attribute by the concatenation of its set of values, then this set is transformed into either an array or a list. It could be also the case when a compound attribute is replaced by the concatenation of its components.

3.3.2. Optional attributes

It is rare in practice that all the information stored in a database is complete. There are different kinds of incompleteness. Among them, we are interested here by the missing information

that is generally of two categories namely *unknown information,* for example, if the name of the course that the student is taking is applicable but unknown at the present time, and *inapplicable information,* for example, if the student is not married or did not have another diploma. This missing information is modelled by a special value called *NULL* value.

In the context of this work, the problem could be of two kinds: attributes that are declared mandatory and those which are declared optional (NULL). For the first type, analysis could give hints about the presence of a special value that simulates the fact that this attribute is not really mandatory; for the second type, we will have just the opposite problem, the field is declared optional but all the records seem to have a value for that field, so, maybe it is in fact a mandatory attribute and, for some obscure reasons, the programmer has preferred to declare it as optional.

3.3.3. Attribute aggregates

We talk about attribute aggregates when a compound attribute is replaced by its components. As a consequence, we have a certain number of attributes which seem to be independent but are in fact just components of one unique attribute.

3.3.4. Multivalued attributes

The task here is to recover multivalued attributes from single-valued ones for which the analysis has shown a kind of repeating structure.

These single valued attributes were obtained by applying a transformation which replaces a multivalued attribute by the concatenation of its values.

3.3.5. Multiple-domain attribute

An attribute could be used as a container, a further analysis could show that this attribute includes values of different types.

For some reasons of optimisation, an attribute could contain several different attributes issued most of the time from different tables.

3.3.6. Secondary identifiers

The secondary identifier of table is not declared or not supported by the DBMS.

3.3.7. Identifiers of multivalued attributes

It concerns multivalued compound attributes for which the identifier was not declared.

3.3.8. Foreign keys

Foreign keys express the interrelational constraints between key attributes in two relations. If a set of attributes in a relation schema is specified as a foreign key referencing the primary key of another relation schema, or if there exists hints about this property, then the values of that set in the database must satisfy the foreign key constraints i.e. the values of the set have to be a subset of the values of the primary key being referenced.

3.3.9. Sets behind arrays...

The DBMSs do not offer the possibility to implement sets which are collections of unordered and distinct values, instead they offer other means to represent the repeating values that is more complex than sets.

In fact, arrays is the main option offered by the DBMSs and can be defined as collections of ordered and non-distinct values with empty cells.

3.3.10. Functional dependencies

Functional dependencies generalise the notion of keys and represent the most common data dependency that arises in practice.

Unfortunately, many relational DBMSs do not provide much support for specifying functional dependencies, which remain implicit in the extensions of the database.

Besides that, some functional dependencies could be lost because of the unnormalized structures. The objective will be to recover them.

3.3.11. Existence constraints

We have four kinds of existence constraints to be recovered: coexistence, exclusive, exact-one and at-least-one.

- Two optional attributes are said to be coexistent if they both have a value or are null together.
- Two optional attributes are said to be exclusive when at most one of them has a value.
- Two optional attributes are related by an exact-one constraint if exactly one of them has a value (at any time).
- Two optional attributes are related by an at-least-one constraint if at least one of them has a value (at any time).

The existence constraints give hints about subtype implementation.

3.3.12. Exact min/max cardinality of attributes and relationship types.

Multivalued attributes declared as arrays have a maximum size that can be limited to a constant. The value of this constant could be an implementation constraint that is independent of the property of the problem. When it is the case, the maximum cardinality of an attribute or a relationship could be relaxed to N instead of the value specified.

The minimum size is under the responsibility of the programmer, the minimum cardinality could be restricted.

3.3.13. Redundancies

Redundancies could be included for performance reasons.

In order to normalise the schema, it is important to detect these redundancies.

3.3.14. Enumerated value domains

We are in presence of enumerated domains when the attributes draw their values from a set of predefined values. It is important to recover this set.

3.3.15. Constraints on value domains

The domain of an attribute is a set of values associated with that attribute, frequently there exits restrictions on the values of the domain. For instance, an integer has to be smaller or equal to 100.

3.3.16. Meaningful names

Some applications have been developed without following any conventions as far as the naming is concerned, which leads to meaningless names and sometimes contradictory ones.

In the other hand, some technical constraints or some programming discipline sometimes impose the usage of meaningless names or condensed ones whose meaning becomes difficult to understand.

Chapter 4

Information sources and elicitation techniques[9]

Figure 8 represents a summary of the main information sources and the techniques related to each of them. In most projects, different sources could be analysed and even coupled to recover the implicit constructs.

4.1. Generic DMS code fragments

SQL code sections such as check/assertion, predicates, triggers and stored procedures could monitor the behaviour of a database. Thus, their exploration can give information of prior importance for the recovery of the structures and constraints they intend to express. We will describe each of them in some detail.

Note that the lack of standard methodology for coding a specific constraint, make the analysis of the generic DMS code fragments difficult.

- Stored procedures: "A stored procedure is a certain piece of code (the procedure), consisting of declarative and procedural SQL statements stored in the catalog of a database that can be activated by calling it from a program, a trigger or another stored procedure" [Van der Lans00].

  A stored procedure is thus a set of SQL statements that are declarative such as *CREATE*, *UPDATE* and *SELECT*, with possibly some procedural statements such as IF-Then-Else. It is executed only when it is called. It is stored in the catalog (a catalog is a set of tables in which all database objects are described).

- Check predicates: The aim of a check predicate is to specify a constraint that has to be verified at any time by the tuples of a table. It is evaluated each time a record is added, updated or removed from the table.

- Assertions predicates: An assertion is a constraint that normally involves several tables.

- Triggers: "A trigger is a piece of code consisting of procedural and declarative statements stored in the catalog and activated by the DBMS if a specific operation is executed on the database and only then when a certain condition holds" [Van der Lans00].

[9] This chapter is derived from [Hainaut98]

Note that triggers can not be called explicitly (as it was the case for the stored procedures), they are called by the DBMS itself, they are executed when an interactive user or stored procedure performs a specific operation such as adding a new row or deleting a row.

4.2. HMI procedural fragments

The validation of some integrity constraints can be implemented by the procedures that monitor the user-program dialogs. These procedures are triggered by interface events or by the updates of the database.

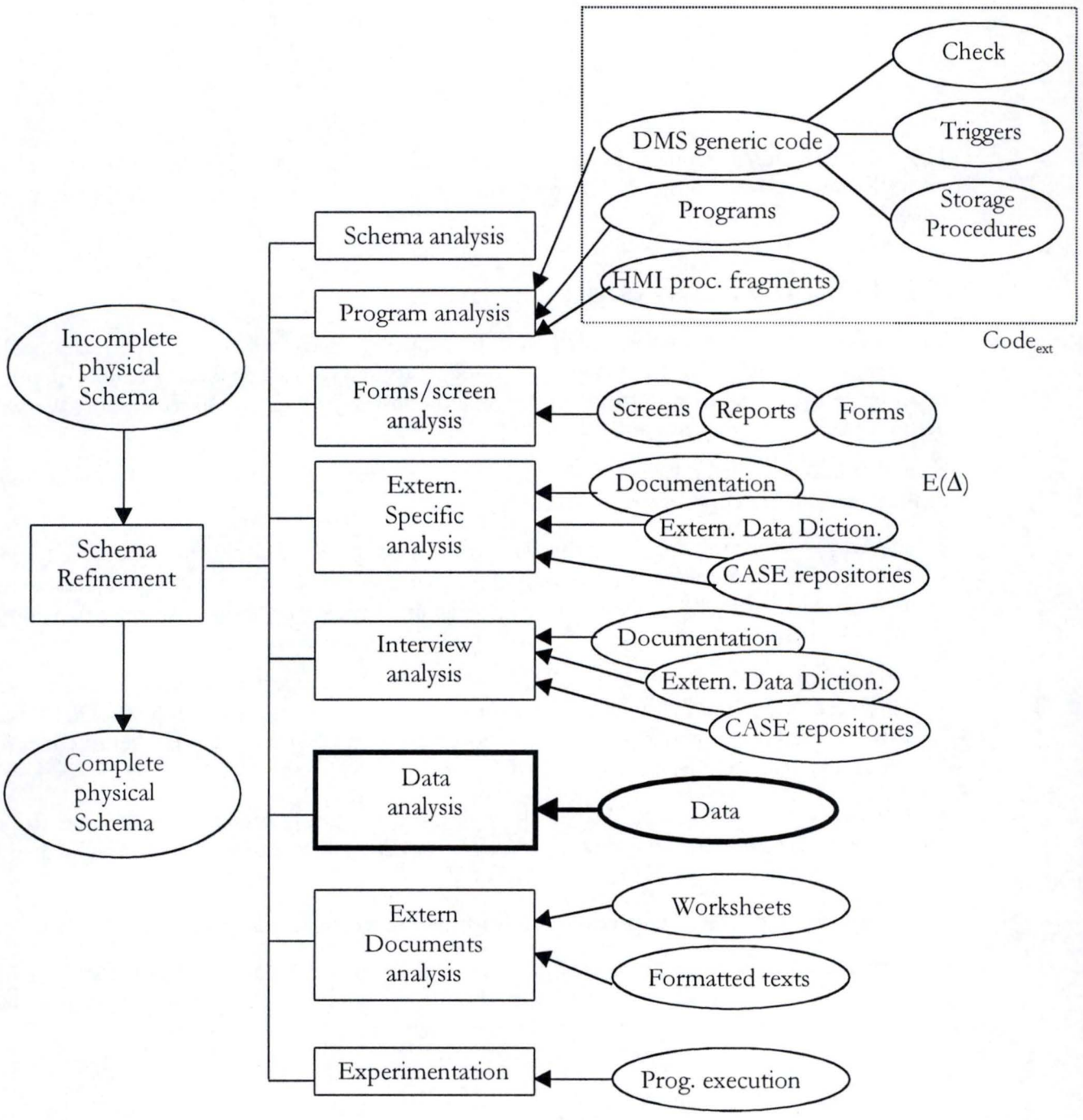


Figure 8-The main information sources and techniques of elicitation.

4.3. Current documentation

In the case that the documentation exists, and even if it is partial, obsolete or incorrect, it can bring some useful information about the structure of data and constraints among them.

4.4. Non-database sources

When data is implemented with general purpose software such as spreadsheets and word processors or when large text databases are implemented according to representation standards such as HTML, it is interesting to analyze these parts.

4.5. Technical/physical constructs

A correlation could exist between logical constructs and their technical implementation. If it is the case, important hints could be given about the logical constructs. For example, foreign keys and indexes are highly related and the fact of finding an index can be considered as an evidence of the existence of a foreign key.

4.6. Names

The interpretation of names, when the variables or the attributes have meaningful ones can bring information about the meaning of the object or about its purpose. The structure of data can also be derived from the analysis of names.

In addition, Synonyms (the same object is referred with different names) and homonyms (different objects with the same name) can be detected.

4.7. External data dictionaries and case repositories

Information about data can be derived from in-house-data dictionary systems. Although this information could be incomplete or obsolete, it helps to catch the semantics of data. Case repositories can also be used to get information about data.

4.8. Program execution

Examining the dynamic behavior of programs working on data could provide the structure and constraints on that data.

4.9. Domain knowledge

The domain knowledge represents the starting point of any reverse engineering project. Indeed, the process of reverse engineering is applicable to systems belonging to different application domains, where each domain has its own specificities. We can say that the process of reverse engineering is domain-dependent, thus, the need to get knowledge on the application domain before starting a reverse engineering process. This knowledge could be derived, among others, from the interviews of analysts, developers and users.

Taking into account the application-domain will help resolving specific problems and enhance the quality of the process of reverse engineering.

In this field we can find different researches such as KBRE project and the work of Jean-Marc Debaud [Debaud96].

The KBRE project (knowledge Based Reverse Engineering of legacy Telecom Software) [TIM 97] represents the domain expert's knowledge within a knowledge-database. The first layer of this base includes the description of objects of interest (without the details). It is obtained from interviewing the users and examining the documentation. They argue that this layer will help in discovering the conceptual problems and (or) the terminology of the domain. Other layers are included such as:

1. The one concerned with the specializations.
2. The one concerned with the representations of the concepts on the source code.

The work of Jean-Marc Debaud [Debaud96] models the application domain and takes it as an input for the reverse engineering process. He considers that the description of the domain provides a set of constructs to search in the code.

4.10. Screen/form/report layout

A screen, a form or a structured report can be considered as derived views of data [Hainaut98]. Indeed, forms gather structured information and constraints among data.

Forms are the most widely used formal communication objects within the organizations. It seems natural that the collection of an organization's forms has to be considered in the process of reverse engineering.

A form can be defined as "Any structured collection of variables (form fields) which are appropriately formatted to communicate with the database, especially for data retrieval and data display" [Mfourga 97].

Forms are generally designed with the aim to be user-friendly. As a consequence they may not mirror the structure of the underlying database, that is the reason according to Mfourga of the necessity of abstraction in order to make the components of the form, their relationships and the constraints among data explicit. He argues that some information such as cardinality constraints, functional dependencies, and existence dependencies could be derived. This information is provided by a phase called by the author the Dynamic Analysis. He derives the structural components and their relationships during the Static Analysis where the intention and the layout are explored.

4.11. Data

The main goal of database design is the storage and manipulation of data which contain useful domain semantics. In the relational model, for instance, the semantics is expressed through the constraints and most of relational DBMSs do not provide support for specifying a certain range of them. As a consequence, to recover them, one do not have to search in the schema but in the extensions that could exhibit some properties such as uniqueness, inclusion. This information could be used to confirm or disprove structural hypotheses; but data could also be used to discover some rules that govern data.

From data analysis, we can derive information about implicit constructs such as functional dependencies. The problem of data dependency was addressed by the specialists of database and by the specialists of an emerging field called *data mining*.

Data mining is the search for valuable information in large volumes of data [Weiss98]. It draws its principles from databases, machine learning and statistics. One can divide the generic types of data-mining problems into two categories: *prediction* and *knowledge discovery* [Weiss98]. Prediction problems are described in terms of specific goals, which are related to past records with known answers. These are used to project to new cases. Knowledge discovery is considered as the process of nontrivial extraction of implicit, previously unknown and potentially useful information from data. Knowledge discovery problems usually describe a stage prior to prediction, where information is insufficient for prediction. Some authors consider *knowledge discovery* closer to decision support than to decision making. We will be interested here by the knowledge discovery category. It uses different techniques to extract information from data, for further details see [Weiss98]. In this work, we will retain the association rules technique and among all the researches done in this area, the work of Agrawal [Agrawal94] will hold our attention. The problem of association rules was introduced in [Agrawal93] for sales transaction databases. The association rules identify sets of items that are purchased together with other sets of items. For example, an association rule may state that 95% of customer buy butter and bread together or 60% of customers that buy butter and bread buy also chocolate. Note that association rules do not try to characterise the content of the database as a whole. Instead of that they try to find relations between the individual records or relation between sets of records. The number of association rules is usually large. A user should not be provided with all of them, but rather with the original, novel and interesting ones[kryszkiewicz98]. The interestingness of a rule is expressed by some qualitative measure. The formal definition of association rules will be as follows[Agrawal94]:

Let $I=\{i_1, i_2, \ldots, i_m\}$ be a set of distinct literals, called items. In general, any set of items is called an itemset. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. An association rule is an expression of the form $X \Rightarrow Y$, where $X \neq \varnothing$, $Y \subset I$ and $X \cap Y = \varnothing$. X is called the *antecedent* and Y is called the *consequent* of the rule. The intuitive meaning of such a rule is that objects in the database which are covered by Y tend to contain objects covered by X. Statistical significance of an itemset X is called the support and is denoted by sup(X). Sup(X) is defined as the number of transactions in D that contain X. Statistical significance (*support* also called *frequency*) of a rule $X \Rightarrow Y$ is denoted by $\sup(X \Rightarrow Y)$ and defined as $\sup(X \cup Y)$ i.e. the percentage of transactions that contain both X and Y, formally, $\sup(X \Rightarrow Y) = Pr(X \cup Y)$, where Pr stands for percentage. Additionally, an association rule is characterised by *confidence* (also called *reability*), which expresses its strength. The confidence of an association rule $X \Rightarrow Y$ is denoted by $conf(X \Rightarrow Y)$ and defined as the ratio $\sup(X \cup Y) / \sup(X)$. His algorithm aims at producing all the rules that have a support greater than some user specified minimum support (minsup) and confidence not less than a user specified minimum confidence (minconf).

The main difference between functional dependencies and association rules resides in the fact that the FDs are rules that require strict satisfaction i.e. support = confidence = 100 % whereas association rules are probabilistic in nature. Indeed, the presence of a rule $X \Rightarrow Z$ does not mean that $X + Y \Rightarrow Z$ holds because the second one may not have the minimum support, but in the area of databases the second rule necessarily holds (augmentation rules[10]); the same thing will be for the transitivity[11]. In fact, in the area of databases[12]:

When $X \rightarrow Y$ holds and $Y \rightarrow Z$ holds then $X \rightarrow Z$ holds by transitivity.

---

[10] See chapter five.

[11] See chapter five.

[12] When dealing with the area of database, we will adopt the symbol "$\rightarrow$" instead of "$\Rightarrow$".

Which is not the case for association rules. In fact, $X \Rightarrow Z$ could not hold because it does not have the minimum confidence.

In the database field, one can find the work of Castellanos, Mannila, and Li [Li93] who has studied the Multivalued dependencies.

The work of Mannila [Mannila94] addresses the functional dependency inference problem[13]. Several algorithms are presented (6 algorithms). The complexity of each one is studied (for more details see [Mannila94]). The first algorithm with a complexity of $O(n^2 2^n p \log p)$ where n is the number of attributes and p is the number of tuples tries to infer a cover[14] of dep(r) where dep(r) represents the set of all functional dependencies holding in a relation r.

The second algorithm do the computation of lhs[15] by pairwise comparison of tuples. Dependency inference using transversal of hypergraphs is the base of two of his algorithms. His fifth algorithm is a sort-based one: it is based on the idea of repeatedly sorting the rows of the relation w.r.t. different orderings of the attributes and computing the lhs for dep(r) by consecutive sorts. The last algorithm computes the lhs beginning with a sample, this lhs will be refined later using the whole relation.

As a summary we can say that he has shown that the functional dependency inference problem is exponential in the number of attributes. He tried to develop algorithms with better best case behavior. The common base of all the algorithms is that for a given rhs[6], he generates the set of possible lhs. He stresses on the number of operations.

In the work of Castellanos [Castellanos], one can find algorithms to extract functional and inclusion dependencies. She focuses on the minimisation of disk accesses and not in the number of operations as it was the case of Mannilla.

Data dependencies are extracted by analyzing the extensions of a database. Her approach is the following:

1. If some semantic constraints are specified in the DDL, they are taken as the initial set of dependencies.
2. The extensions of the relations are analyzed for additional dependencies, the dependencies derived by implication rules are not checked by data analysis.
3. Involvement of the user.

In this approach, all the possible rhs of a given lhs are generated (while Mannila's work generated the lhs for a given rhs). Only the elementary functional dependencies are considered. For the inclusion dependencies (INDs), two algorithms are proposed. She uses some heuristics to reduce the set of possible INDs:

- The attributes in the lhs and those in the rhs of an IND must have the same type.
- She considers only the domains used for identifiers.
- She imposes a limit on the length of the INDs considered.
- She starts with unary INDs and she discards all possible non unary INDs for which it does not exist a unary IND for each pair of the corresponding attributes.
- She discards all the ternary INDs for which there does not exist a binary IND for each pair any of the corresponding pairs of attributes.

---

[13] See chapter five.

[14] See chapter five.

[15] X is the lhs of the functional dependency $X \rightarrow Y$ and Y is its rhs.

Note that data dependencies are not the only information that can be derived from data. Indeed, existence constraints, optional fields are some examples of what one can derive from data.

Note that data is the main source of information that will be used in this work. From the approach of Mannilla, we will retain the use of samples when dealing with the whole database. We will follow castellanos in generating the rhs for a given lhs.

4.12. Application programs

Programs that manage, use and transform data could be used to induct information on the structural properties of these data. In the domain of program understanding, we are provided with different tools and techniques of analysis that we will try to describe in some details below.

Because structures and constraints could be encapsulated in a certain number of programs, the program understanding can be useful to catch these structures and constraints and help the process of reverse engineering.

The reason of using programs to express constraints and structures are numerous: it could be because the developer wanted to meet genericity, simplicity or efficiency requirements or it could be because of poor programming practices…

- Dataflow analysis: "Examining in which variables data values flow in the program can put in light structural or intentional similarities between these variables" [Hainaut98]. Where the word flow has to be understood as "if two variables belong to the same graph, at some time, their values could be the same or one of them is direct function of the other." [Hainaut98].
- Dependency analysis: It is a generalization of the dataflow diagram. It is a graph where the nodes are the variables of the program, the nodes are linked with each others (in the case of the existence of a relation between the variables) by edges. These edges can be directed or not. If, in the program, there exists a relation between variable X and variable Y, this relation will be represented on the graph by a path (an edge) linking the node X to the node Y.
- Program clichés analysis: When the pattern (or cliché) for a definite problem was identified, it can be used to search the programs. This enables to locate where problems of this kind are solved.
- Program slicing: Program slicing is a decomposition based on data flow and control flow analysis, it reduces the space of search. Indeed, it starts from a subset of a program's behavior, and produces a reduced program called *slice* that produces the same behavior of interest as the complete program. A *slicing criterion* of a program P is a tuple (i, v) where i is a statement in P and V is a subset of the variables in P. The definition of a slice follows: "A slice S of a program P on a slicing criterion C = (i , v) is any executable program with the following two properties : S can be obtained from P by deleting zero or more statements from P. Whenever P halts on an input I with state trajectory T, then S also halts on input I with state trajectory T"[Weiser84].

Chapter 5

The use of data analysis with relational databases

This chapter will be composed of two sections. The first section is a reminder about relational databases where we will explain the major concepts related to this model. The second section will deal with the recovery of the major implicit constructs and constraints using the technique of *Data Analysis*. For each construct/constraint, we will start by the theoretical background which will be followed by an account for the proposed solution of elicitation. The second section will end with a proposal of a strategy, which aims to organise, as much as possible, the reasoning.

We will use the following notations:

- $A \models F$ means A verifies (or satisfies) F or A logically implies F.
- $A \vdash F$ denotes the fact that we can find in A a proof of F.

5.1. Relational databases

The aim of this section is to give a brief presentation of relational database systems. E.F Codd has introduced the relational model, which is the basis of most database systems, in 1970. The only data structure provided by the model is the relation: it is a formal notion that comes from set theory in mathematics. The mathematical definition does not indicate whether the sets on which the relation is defined, are finite or not, so the relation itself could be finite or infinite. In practice, and more precisely in the field of databases, the relations are necessarily finite whereas the sets could be finite or infinite. We will start by the notion of attributes and domains then we will define some keywords used in the relational data model such as database schema, relation schema, relation instance...This section is based on [Atzeni99], [Levene99], [Elmasri00] and the course of professor Hainaut [Hainaut98].

5.1.1. Basic concepts:

- An *attribute* of a relation is a property of that relation. An attribute is *simple* if it is made of a single attribute otherwise it is *composite*. The *domain* of an attribute indicates the values taken by that attribute i.e. it is a set of values associated with the attribute of interest, it can be seen as giving semantics to the attribute. A domain is *atomic* (or primitive) if its values are nondecomposable. Examples of atomic domain are the domain of all positive integers, the domain of integers between 5 and 500. A domain is *set-valued* if its values are finite sets of atomic domains. Examples of set-valued domains are the domains of finite sets of integers or finite sets of strings.

We can give another definition of domains will as follows:

Let U be a countably infinite set of attributes (A countably infinite set is a set that can be put into one-to-one correspondence with the set of all natural numbers). U is called the universe of attributes.

Let D be a countably infinite set of values. D is called the underlying database domain.

Given an attribute A in U, the domain of A, Dom (A) is a subset of D.

We introduce the notion of *tuple* by saying that a tuple on a set of attributes X is a function which associates with each $A \in X$ a value of Dom (A).

- A *relation schema* [16] R denoted by R(X) consists on the name of the relation R and the set of its attributes $X = \{A_1, A_2, \ldots A_n\}$ where a domain is associated with each attribute. Two notations are available $R(A_1, A_2, \ldots A_n)$ or $R(A_1{:}Dom(A_1)\ldots A_n{:}Dom(A_n))$. For example, if we have (a relation) Players with as attributes set {name, address, age}, the relation schema will be Players(name:char, address:char, age:integer) or Players(name, address, age).

Note that the set of attributes of a relation schema R are denoted by ***schema(R)***, for the above example we will have schema(Players) = {name, address, age}.

- A *database schema* D is a set of relation schemas denoted by $D = (R_1(X_1), R_2(X_2), \ldots R_n(X_n))$, where $\forall\ 1 \leq i \leq n$, $R_i(X_i)$ is a relation schema.
- A *relation instance* (or relation state)[17] r of the relation schema $R(A_1, A_2, \ldots A_n)$ is a set of n-tuples $r=\{t_1, t_2, \ldots t_m\}$. Each *n-tuple* t is an ordered list of n values $t:=<v_1, v_2, \ldots v_n>$, where each value v_i, $1 \leq i \leq n$, belongs to $Dom(A_i)$. The i^{th} value in tuple t will be referred to as $t[A_i]$ and said the *restriction* of t to A_i.
- A *database instance* on a database schema $D = (R_1(X_1), \ldots R_n(X_n))$ is a set of relation instances $d= (r_1, r_2, \ldots r_n)$ where, $\forall\ r_i$, for $1 \leq i \leq n$, r_i is a relation on the schema $R_i(X_i)$.
- The number of attributes is called the *degree* of the relation schema.
- The number of tuples in the relation r is called the *cardinality of the relation* denoted $|r|$ or *card(r)*.
- A *relational database* is a set of relations.

As we have seen it, the structure of the relational model is simple and powerful, but it also imposes a certain degree of rigidity; in fact, the information has to be represented by homogenous tuples of data. In order to represent the non availability of values, a concept of " *null value* " was introduced and it denotes an absence of information.

5.1.2. Relational Algebra

The relational algebra is a collection of operators that was presented by Codd in 1970. The aim of these operators is to derive new structures from one or more existing structures given as input. We will start with the set theoretic operators: union, difference and intersection, after which we will introduce the projection, selection, relational product and join operators.

1. Union: the union of r_1 and r_2 denoted by $r_1 \cup r_2$ or $r_1 r_2$ is defined as:

$$r_1 r_2 = \{t \mid t \in r_1 \text{ or } t \in r_2\}$$

[16] Also called relation *intension*.

[17] It is also called *relation* or *relation extension*.

Intuitively, the union of two relations, r_1 and r_2 over relation schema R, is the set of tuples that are either in r_1 or in r_2

Where r_1, r_2 are relations over relation schema R t is a tuple.

2. Difference: The difference between r_1 and r_2 denoted by $r_1 - r_2$ is defined as:

$$r_1 - r_2 = \{t \mid t \in r_1 \text{ and } t \notin r_2\}$$

Intuitively, the difference between r_1 and r_2 over a relation schema R is the set of tuples that are in r_1 but not in r_2

3. Intersection: it is the set of tuples that are in both r_1 and r_2. Formally:

$$r_1 \cap r_2 = \{t \mid t \in r_1 \text{ and } t \in r_2\}$$

The intersection can also be defined in terms of difference:

$$r_1 \cap r_2 = r_1 - (r_1 - r_2).$$

4. Projection: The projection, $\prod$, of a relation r over R(X) onto a set of attributes Y ($Y \subseteq$ schema(R)) is expressed as:

$$\prod_Y (r) = \{t[Y] \mid t \in r\}.$$

Where t[Y] is the restriction of t to Y.

5. The projection of a relation r over relation schema R onto a set of attributes Y included in schema(R) is the set of tuples resulting from projecting each of the tuples in r onto Y.

6. Selection: It is denoted as $\sigma_{<\text{selection condition}>}(r)$. Where the symbol σ denotes the selection operator and the selection condition is a Boolean expression specified on the attributes of R. It is composed by clauses of the form

<attribute name> <op> <constant>, or <attribute name> <op> <attribute name>

where <attribute name> is the name of an attribute of R, <op> is an operator belonging to the set $\{=, \neq, <, >, \leq, \geq\}$ and <constant> is a constant value from the attribute domain. Clauses can be connected by the Boolean operators AND, OR and NOT.

The selection operator is used to select a subset of the tuples from a relation r over a relation schema R that satisfy the selection condition.

$\sigma_{<\text{selection condition}>}(r) = \{t \mid t \in r \text{ and selection condition on t is true}\}$

7. Relational Product[18] : It creates tuples with the combined attributes of two relations. the relational product, $\otimes$, of relations r_1 over relation schema R_1 and r_2 over relation schema R_2 is a relation r that has one tuple for each combination of tuples, one from r_1 and one from r_2 (i.e. a concatenation of tuples). If r_1 has n_1 tuples and r_2 has n_2 tuples r will have $n_1 * n_2$ tuples. It is a meaningless operation in itself but if combined with a selection that matches values of attributes coming from the component relations, it becomes useful. Because this sequence of Relational Product followed by a selection is frequently used, another operator was created called *join*.

[18] It is refered as Cartesian Product by some authors, but we prefer to follow [Hainaut98] when he refers to this operator as "*Produit Relationnel*" to avoid any confusion with the mathematical notion of Cartesian Product. This operator is also known as *Cross Product* or *Cross Join* [Elmasri00].

8. Join: Given two relation schemas R_1 and R_2, the join $\triangleright\triangleleft$ of two relations $r_1$over relation schema R_1 and r_2 over relation schema R_2 is a relation r over relation schema R. is defined by:

   $r = r_1 \triangleright\triangleleft r_2$

   $r = \{t \mid \exists\, t_1 \in r_1 \text{ and } \exists\, t_2 \in r_2\,,\ t[\text{schema}(R_1)] = t_1 \text{ and } t[\text{schema}(R_2)] = t_2\}$

   Note that schema(R) = schema(R_1) $\cup$ schema(R_2).

   Informally, the join of two relations r_1 over relation schema R_1 and r_2 over relation schema R_2, with schema(R_1) $\cap$ schema(R_2) being the set of attributes X, is the relation containing tuples that are the concatenation of every tuple of r_1 with every tuple of r_2 having the same X-values.

5.1.3. Integrity constraints in Relational Databases

In general, we restrict relations to satisfy certain conditions, called *constraints*. They are the properties that have to be respected by all the instances of the database, these instances will be called correct database instances. Each constraint could be seen as a first-order logic statement that associates the value true or false with each instance. As a first approach we can say, following [Atzeni99] that we have two categories of constraints:

1. *Intra-relational constraint*: it is a constraint defined with regard to a single relation. It includes some sub-categories such as:
   - A tuple constraint: is a constraint that can be evaluated on each tuple independently from the others.
   - Domain constraint: it imposes a restriction on the domain of the attributes.
2. *Inter-relational constraints*: when more than one relation is involved.

Another kind of classification of constraints is proposed in the literature: Static Constraints and Dynamic Constraints.

We deal with a *Static Constraint if* "Its satisfaction in a database can be checked by examining the current database state" [Levene99] whereas the satisfaction of *Dynamic constraints* could be checked by investigating two successive database states. The static constraints include data dependencies, domain dependencies and cardinality constraints that will be explained in further sections.

A sub-type of dynamic constraints is the state transition constraints [Levene99]. By example, a constraint about salaries of employees of a firm stating that the salary increases of one per cent each year.

5.1.4. Data dependencies

Constraints that depend on the equality or inequality of values in tuples of relations such as functional and inclusion dependencies are called data dependencies. We will start by functional dependencies then we will be interested in referential constraints and inclusion dependencies.

A. Functional dependencies

A *functional dependency* F over R (FD) has the form F: X → Y where X and Y are simple or composite attributes and X $\subset$ schema (R) and Y $\subset$ schema (R). X is called the *determinant* or the left hand side (*lhs*) of the functional dependency F and Y is called the right hand side (*rhs*) of the functional dependency F.

A FD F: X→ Y is *satisfied* in a relation r over a relation schema R if whenever two tuples in r have equal X-values they also have equal Y-values. As a consequence, every X-value in r has only one corresponding Y-value. The formal definition of satisfaction of a FD in a relation follows.

A FD F: X→ Y is satisfied in a relation r over a relation schema R denoted by $r \models X \rightarrow Y$:

If $\forall (t_1,t_2) \in r$, if $t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]$. Where t[X] is the projection of a tuple t on a set of attributes X and represents the restriction of t on X and $X,Y \subseteq$ schema(R). That means,the values of the X component functionally determine the values of the Y component.

We will try here below to present the different types of functional dependencies that one can encounter. After that, we will have a glance at the inference rules for functional dependencies. These rules will help us, when dealing with the retrieval of all the functional dependencies holding in a relation or a database, to reduce the space of search. Indeed, let A, B, C be attributes of a relation schema R, finding out that A → B and B → C holds in R will permit us to infer whithout checking the instances of the database that A → C holds also in the relation. Then we will investigate the theory about the implication problem for FDs and minimal covers of FDs.

We have different types of FDs, their definitions is given below:

- A FD X→Y is *trivial* if $Y \subseteq X$.
- A FD X→Y is *standard* if $X \neq \emptyset$.
- A FD X→Y is *degenerated* or non standard if $X = \emptyset$ (rare in practice).
- A FD X → Y is said to be *elementary* if and only if Y is a single attribute, $\neg \exists X' \subset X$ such that X' → Y (X is minimal), and X → Y is non trivial.

Before presenting the inference rules for FDs, we will introduce some concepts. An *inference rule* is a rule which allows us to derive an integrity constraint from a given set of integrity constraints. More precisely, an inference rule is a sentence of the form: *if* certain integrity constraints can be derived from the given set of constraints *then* we can derive an additional constraint. The *if* part is called the hypothesis of the rule and the *then* part of the rule is called its conclusion.

An *axiom* is an inference rule with an empty *if* part, that is, the additional constraint can be derived unconditionally. An axiom system for a class C of integrity constraints is a set of inference rules with respect to C. A class of integrity constraints refers to a particular set of integrity constraints over a given relation schema or database schema.

a. Inference rules for FDs

Let F be a set of FDs over schema R, we have the following inference rules for FDs.

1. Reflexivity: if $Y \subseteq X \subseteq$schema(R), then $F \vdash X \rightarrow Y$.
2. Augmentation: if $F \vdash X \rightarrow Y$ and $W \subseteq$ schema(R), then $F \vdash XW \rightarrow YW$.
3. Transitivity: if $F \vdash X \rightarrow Y$ and $F \vdash Y \rightarrow Z$, then $F \vdash X \rightarrow Z$.

These first three rules are known as Armstrong's axiom system

4. Union: if $F \vdash X \rightarrow Y$ and $F \vdash X \rightarrow Z$, then $F \vdash X \rightarrow YZ$.
5. Decomposition: if $F \vdash X \rightarrow YZ$, then $F \vdash X \rightarrow Y$ and $F \vdash X \rightarrow Z$.
6. Pseudo-transitivity: if $F \vdash X \rightarrow Y$ and $F \vdash YW \rightarrow Z$, then $F \vdash XW \rightarrow Z$.

The reflexive rule states that a set of attributes always determines itself or any of its subsets, which is obvious. Because the reflexive rule generates dependencies that are always true, such dependencies are called trivial. The augmentation rule says that adding the same set of attributes to both the left – and right – hand sides of a dependency results in another valid dependency. According to third rule, functional dependencies are transitive. The decomposition rule says that we can remove attributes from the right-hand side of a dependency ; applying this rule repeatedly, we can decompose the FD $X \rightarrow \{A1, A2, \ldots, An\}$ into the set of dependencies $\{X \rightarrow A1, X \rightarrow A2, \ldots, X \rightarrow An\}$. The union rule allows us to do the opposite; we can combine a set of dependencies $\{X \rightarrow A1, \ldots, X \rightarrow An\}$ into the single FD $X \rightarrow \{A1, A2, \ldots, An\}$.

It was established in the work of Mannila [Mannila94] that the complexity of the dependency inference as a function of the number of rows requires in the worst case Ω[19] (p log p) steps for two attribute relations with p rows.

Note that two types of research are undertaken in the field of functional dependencies. The first kind deals with the satisfaction problem of functional dependencies. The second research deals with the implication problem for functional dependencies, which will be defined below.

b. The implication problem for functional Dependencies

It is the problem of deciding given a set of FDs F over a relation schema R, and a single FD $X \rightarrow Y$, whether $F \models X \rightarrow Y$ (i.e. the FD $X \rightarrow Y$ is a consequence of F). It was shown [Levene99] that this problem is equivalent to that of deciding whether Y is in the closure of X (i.e. $Y \subseteq X^+$ holds). Note that the closure of a set of attributes C is defined by

$C^+ = \cup \{Y \mid F \vdash C \rightarrow Y$ using Armstrong's axiom system$\}$ where Y is a set of attributes. This problem could be considered as a membership problem that will not be addressed in this work.

c. Minimal covers for sets of functional dependencies

In this section, we start by giving some preliminary definitions. A set of functional dependencies F is *covered by* a set of functional dependencies G. if every FD in F is also in G+; that is, if every dependency in F can be inferred from G. We can determine whether G covers F by calculating X^+ w.r.t. G for each FD $X \rightarrow Y$ in F, and then checking whether this X^+ includes the attributes in Y. If this is the case for every FD in F, then G covers F. Two covers are said to be equivalent, if $F^+ = G^+$ that means the set of FDs derived from F can also be derived from G and conversely. Note that the concept of a cover of a set of FDs is an equivalence relation in the sense of set theory (reflexive, symmetric and transitive).

Before introducing the different types of covers, we will introduce the concept of minimal FDs [Elmasri00].

As set of functional dependencies E is *minimal* if it satisfies the following conditions:

1. Every dependency in E has a single attribute for its right-hand side.
2. We cannot replace any dependency $X \rightarrow A$ in E with a dependency $Y \rightarrow A$, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to E.
3. We cannot remove any dependency from E and still have a set of dependencies that is equivalent to E.

We have three types of cover for FDs:

---

[19] Where the symbol Ω denotes the complexity in the worsecase.

1. F is *non redundant* if $\neg \exists$ a cover G of F such that $G \subset F$.
2. F is *minimum* if $\neg \exists$ a cover G of F such that G has fewer FDs than F. a minimal cover could also be defined as follows: A minimal cover of a set of dependencies E is a minimal set of dependencies E_{min} that is equivalent to E [Elmasri00].
3. F is *optimum* if $\neg \exists$ a cover G of F such that G has fewer attributes than F.

It can be easily verified that a minimum cover is non redundant. Note that minimising the cover of F (a given set of FDs) permits to reduce the time to compute the closure of a set of attributes. Note also that our interest on the notion of minimal covers and minimal FDs could have many justifications. One of the reasons, is that, when the question of the user consists on finding all the FDs holding in a relation or a database, it is useful to have some artefacts to reduce the complexity of the problem and to limit the search to the most interesting FDs i.e. minimal FDs.

B. Referential constraint

The r*eferential integrity constraint* is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation.

A foreign key between a set of attributes X of a relation schema R_1 on another relation schema R_2 is satisfied if the values in X of each tuple of the instance of R_1 appear as values of the (primary) key of the instance of R_2. This constraint will be studied with more detail in the second section.

C. Inclusion dependencies

Inclusion Dependencies (INDs) generalise the notion of referential integrity constraint. It is a constraint specifying that the values of a given subset of attributes of a relation must be a subset of a given set of attributes of a relation that could be similar or different of the first one. Note that INDs are different from functional dependencies, in that they can express interrelational constraints. INDs can be defined as follows:

Let R_1, R_2 be two relation schemas of a database schema. Let X, Y be sequences of attributes such that $X \subseteq \text{schema}(R_1)$ and $Y \subseteq \text{schema}(R_2)$ and $|X|=|Y|$. An IND is an expression of the form: $R_1[X] \subseteq R_2[Y]$. We say that an IND $R_1[X] \subseteq R_2[Y]$ is *satisfied* in a database d which is denoted as $d \models R_1[X] \subseteq R_2[Y]$, if $\forall\ t_1 \in r_1\ \exists\ t_2 \in r_2$ such that $t_1[X] = t_2[Y]$. Where $r_1, r_2 \in d$ and are relations over respectively R_1 and R_2.

We have a very important class of INDs that we will introduce, it is that class of INDs where the attributes in the right hand side are keys. An IND $R_1[X] \subseteq R_2[Y]$ is said to be *key-based* if Y is a key for R_2. An IND is *trivial* if it is of the form $R[X] \subseteq R[X]$. A set of INDs I is said to be *circular* if either a non trivial IND $R[X] \subseteq R[Y] \in I$ or it exists m distinct relation schemas $R_1, R_2, \ldots, R_m$ with $m > 1$ such that I contains the following INDs:

1. $R_1[X_1] \subseteq R_2[Y_2]$.
2. $R_2[X_2] \subseteq R_3[Y_3]$ and ... and $R_m[X_m] \subseteq R_1[Y_1]$.

An IND is *unary* if $|X| = 1$. An IND $R[X] \subseteq S[Y]$ is *typed* if X=Y. A set of INDs is said to be *non circular* if it is not circular.

Note that Foreign keys can be seen as a subclass of INDs called *key-based INDs* and that the only INDs known by SQL-92 are foreign keys.

The inference rules for INDs are expressed as follows:

Let I be a set of INDs over a database schema $D=\{R_1, R_2 \ldots R_n\}$. Let R be a relation schema over D.

1. Reflexivity: if $X \subseteq$ schema(R) then $I \vdash R[X] \subseteq R[X]$.
2. Projection and permutation:
   if $I \vdash R_1 [X] \subseteq R_2 [Y]$ then $I \vdash R_1 [Ai_1, Ai_2, \ldots Ai_k] \subseteq R_2 [Bi_1, Bi_2, \ldots Bi_k]$.

   Where

   $X= <A_1 \ldots A_m>$ a sequence of distinct attributes and $X \subseteq$ schema(R_1).

   $Y=<B_1, \ldots B_m> \subseteq$ schema(R_2).

   $i_1 \ldots i_k$ is a sequence of distinct natural numbers from $\{1 \ldots m\}$.
3. Transitivity: if $I \vdash R_1 [X] \subseteq R_2 [Y]$ and $I \vdash R_2 [Y] \subseteq R_2 [Z]$ then $I \vdash R_1 [X] \subseteq R_3 [Z]$.

We will introduce below the implication problem for INDs where the results of the work of Levene [Levene99] will be presented:

- The implication problem for INDs is PSPACE –complete.
- The implication problem for non circular INDs is NP-complete.
- The implication problem for unary INDs is linear-time in the size of the input set of unary INDs.
- The implication problem for typed INDs is polynomial-time in the size of the input set of typed INDs.

5.1.5. Keys

We will deal here with the concepts of candidate keys, primary keys, alternate keys, superkeys and antikeys.

Since the body of a relation is a set, which by definition does not contain duplicate elements, it follows that no two tuples of a relation can be duplicates of each other. Let R be a relation schema with attributes $A_1, A_2, \ldots A_n$. The set of attributes $C = (A_i, A_j, \ldots, A_k)$ of R is said to be a *candidate key* (also referred as secondary identifier) of R if and only if it satisfies the two independent properties:

1. Uniqueness
   At a given time, no two distinct tuples of R have the same value for $A_i, A_j \ldots A_k$.
2. Minimality
   None of the attributes of C could be discarded without destroying the uniqueness property.

Since functional dependencies generalise the notion of keys, we can give another definition of candidate keys as follows: A set of attributes $C \subseteq$ schema (R) is a candidate key for R with respect to a set of functional dependencies F over R if it satisfies two properties:

1. Uniqueness: $C^+ =$ schema (R) where C^+ represents the closure of a set of attributes defined by $C^+ = \cup \{Y \mid F \vdash C \rightarrow Y\}$ and Y is a set of attributes.

2. Minimality: no subset of C is a key for R with respect to F.

Note that every relation has at least one candidate key. For a given relation, one of the candidate keys is called *primary key*, where null values are forbidden and the others are referred as *alternate keys*. A set of attributes C $\subseteq$ schema (R) is a *superkey* for R with respect to a set of functional dependencies F over R, if C satisfies the uniqueness condition but not necessarily the minimality condition. An *antikey* for a relation schema R with respect to F over R is a maximal set of attributes included in R that is not a superkey for R with respect to F.

Note that an attribute in schema (R) that belongs to at least one candidate key for R with respect to F is called a *prime attribute*; an attribute that is in schema (R) which is not prime is called *nonprime*. A key is said to be *simple* if it consists of a single attribute; otherwise it is said to be *composite*. *Surrogates keys* (i.e. technical identifiers) are keys with no intrinsic meaning and are normally simple attributes.

5.1.6. Normal Forms:

A. First Normal Form (1NF)

A relation schema R is said to be in 1NF if all its attributes are simple (atomic) and mono valued. It is also called a normalised schema or a flat schema. Conversely, R is said to be non-1NF (N1NF) when some of its attributes are composite and/or multivalued. The different other kinds of normal forms were introduced in the relational theory to solve the anomalies and redundancies that could be present in 1NF relations.

B. Second Normal Form (2NF)

A relation schema is in 2NF w.r.t a set F of FDs if:

- It is in 1NF.
- For every FD: X→ A non trivial, either X is a superkey or at least one of the attributes in X is non prime or A is a prime attribute (i.e. no partial dependency).

C. Third Normal Form (3NF)

A relation schema R is in 3NF if:

- It is in 2NF.
- It does not exist any non-prime attribute that depends on other non prime attributes (i.e. no transitive dependency). In other words: R is 3NF ⇔ each attribute that is transitively dependent on a key, this attribute is prime.

D. Boyce-Codd Normal Form (BCNF)

A relation schema R is in BCNF if:

- It is in 1NF.
- Every determinant of an FD is a key.

5.2. Elicitation of implicit constructs

This section will deal with the recovery of a major part of implicit constructs and constraints mentioned in chapter three. The implicit constructs and constraints to be tackled here are functional dependencies, foreign keys, optional attributes, cardinality constraint, fine grained structure of attributes, attribute aggregates, multivalued attributes, multiple-domain attributes, candidate keys, sets behind arrays, existence constraints, redundancies, enumerated value domains and constraints on value domains.

Note that we have discarded some of the implicit constructs and constraints cited in chapter three which is based on the work of Professor Hainaut [Hainaut98]. The reasons underlying this, are that the author is not only interested in relational database but also in other kinds of databases such as Cobol. As a consequence, some of the implicit constructs and constraints cited in chapter three are not applicable for relational databases. Besides this, the approach of the author, is more general than the approach adopted in this work. We mean by that, we are restrained in this work to the use of a special technique, which is Data Analysis.

As already said, a range of techniques is available, the technique used in this work is Data Analysis.

The organisation of this section will be as follows: for each construct/constraint, we elaborate a theoretical reminder followed by the proposed solution and when necessary the difficulty of the problem will be discussed and the proposed heuristics to reduce this difficulty will be presented. The proposed solutions are problem dependent, we mean by that, we do not try to find general and global solutions for a construct or a constraint, instead of that we are guided by the questions of the user for which we try to find out suitable resolutions. As it is the case, the solution could be formed by a set of SQL queries combined or not with a procedural part.

The approach adopted in this work, aim to be as practical as possible in the sense that we are guided in the performed search by the questions of the user which reflect practical problems encountered in real reverse engineering projects. That is one of the reasons of the use of heuristics in this work. Besides that, Data Analysis is a costly technique and any attempt to limit the space of the search, so the expensiveness of this technique is worth so much.

We will end this section by establishing a strategy for the recovery of implicit constructs and constraints that one can encounter in relational databases, at this stage the stress will be put on the role of data analysis. By strategy one have to understand two things: a strategy organising the different components inside the data analysis technique and a strategy where the data analysis is seen as a whole among the other techniques. In this chapter, we will deal only with the first kind of strategy whereas the second kind of strategy will constitute the essence of chapter 6.

Before tackling the essence of this work, we would like to raise a standpoint about which we have to be conscious from the beginning to the end of this section. Although Data Analysis is a very significant technique that could give hints about some situations that are impossible to get when using the other techniques, the results provided by this technique have to be always moderated. Indeed, as the basis of the data analysis technique is data instances, which mirror the current state of the database, any result that is provided by this technique has to be temperate. We mean by that, the results are closely dependent of the current state of the database and if the instances change the results are very likely to change.

In our approach data Analysis has to be registered inside an incremental process i.e. the results provided today are hypotheses for the search to undertake in the future.

Before starting this section, we would like to point that in this work Data Analysis is used for proving/disproving any hypothesis formulated about an implicit construct/constraint or for discovering them.

5.2.1. Data dependencies

Constraints such as functional and inclusions dependencies, which depend on the equality or inequality of values in tuples of relations in a database, are called *data dependencies*. The theory of data dependencies has been central in the research area of relational databases, since it deals with the foundations of the integrity part of the relational model and generalises the fundamental notions of keys and foreign keys [Levene99].

The data dependencies addressed in this work are functional dependencies and foreign keys.

A. Foreign keys

We will start with an introduction on the theory related to the concept of foreign keys. This introduction will permit us to discover the large panoply of foreign keys that one can confront and will help us to moderate our interpretation of the results provided by the process of elicitation.

a. Introduction

A foreign key is an attribute (or a combination of attributes) of a relation, Ro, referred as the origin of the foreign key whose values have to match those of the primary key of a relation Rd called the target relation. Note that Ro and Rd are not necessarily distinct and that the foreign key and the corresponding primary key have to be defined on the same domain. Unfortunately, it exists a variety of foreign keys where most of the time the definition given above is not respected.

We propose to have a closer look to the different kinds of foreign keys that one can encounter in real situations. The panoply of foreign keys was established by professor Hainaut [Hainaut97] and could be divided according to him into five groups namely Standard foreign keys, Pathological foreign keys, Loose foreign keys, Complex foreign key patterns and Non standard foreign keys. We will explore each group to find out the different sub groups and their characteristics.

i. Standard foreign keys and basic variants

- Standard foreign keys: A foreign key is called standard when its attributes are mandatory, it targets the primary key of a relation and it is defined on the same domain as the primary key of the target relation.
- Optional foreign keys: If the attribute of the foreign key is optional , then this foreign key is called optional foreign key i.e. it has one target or none. When the foreign key is multi-component, all the component attributes are optional and are related by a coexistence constraint (the component attributes are optional and present together or absent).
- Total foreign keys: When an inclusion constraint from the primary key of the target relation to the foreign key holds, we refer to this foreign key as total foreign key.
- Identifying foreign keys: When the foreign key is a candidate key of the target relation.
- Cyclic foreign keys: We have a cyclic foreign key when the source relation Ro, and the target relation Rd of a foreign key are the same.

ii. Non standard foreign keys

- Secondary foreign keys: A secondary foreign key is a foreign key based on the secondary identifier of the target relation.
- Multi-target foreign keys: A multi-target foreign key is a foreign key that has more than one target relation. The target relations can be referenced at the same time.
- Alternate foreign keys: An alternate foreign key is a foreign key that has more than one target but does not reference all of them simultaneously, at the contrary of multi-target foreign keys where all the targets can be referenced at the same time.
- Computed foreign keys: A computed foreign key occurs when a component of the foreign key is an indirect reference to the correspondent component of the key of the target relation.
- Non 1NF foreign keys: It occurs when the attributes of the foreign key are multivalued or compound attributes.
- Partial foreign keys: When the foreign key matches a fragment of the target identifier, we talk about a partial foreign key.

iii. Complex foreign key patterns

- Conditional foreign keys: The reference behaves as a foreign key under certain circumstances, and when the conditions are not satisfied, it has another interpretation.
- Overlapping identifier foreign keys: The foreign key shares some attributes with the identifier of the source relation.

  We have two cases:
  - The attributes of the foreign key are strict subset of the identifier (target key).
  - The intersection between the foreign key and the target key is not empty, and neither the identifier nor the foreign key includes the other one.
- non minimal foreign keys: It happens when the target key of the foreign key is not minimal.
- partially reciprocal foreign keys: It is a way to represent a one-to-one relationship set included into another relationship set.

iv. Loose foreign keys

- Loosely matching foreign keys: It occurs when the domain of the foreign key and the domain of the target-key are not the same. The two keys satisfy the looser rule, which means they are comparable in some way.
- 99 % correct foreign keys: It is when the construct behaves as a foreign key most of the time.

v. Pathological foreign keys

- Transitive foreign keys: A transitive foreign key is the composition of two or more other foreign keys. In most cases, they are not explicit.
- Partly optional foreign keys: Some of the attributes of the foreign key are optional.

- Embedded foreign keys: An embedded foreign key is a foreign key composed of attributes that are components of another foreign key.
- Reflexive foreign keys: It is a valid foreign key but useless.

b. Elicitation

We will examine in this section particularly how data analysis can help us in the process of recovery of implicit foreign keys in relational databases.

Note that Data analysis could be used in several ways. It could be used for proving, disproving foreign key existence or detecting them when we do not have any prior knowledge.

We will first show how data analysis could be used for proving/disproving the existence of foreign keys then we will demonstrate its use for discovering.

i. Use of data analysis for proving/ disproving

If we use data analysis for proving/disproving we could have the situation where for example, if we know that, let us say an attribute A_1 of a relation A is a foreign key targeting the relation $B(\underline{B_1},B_2,B_3,\ldots B_n)$ where B_1 is the primary key of the relation B, B_2 is a candidate key of B.

We have, first of all, to distinguish the different kinds of foreign keys we are looking for.

The first case will be foreign keys having as target a primary key of a relation; we mean that the target of the foreign key is the primary key of another relation or of the same relation. The second case will be foreign keys having as target a secondary identifier of a relation (a secondary identifier is an identifier that could be NULL. In this work we will deal only with the first type of foreign keys.

To recover the first kind of foreign keys, we will have an SQL query like:

```
select count(*)
from B
where A1 not in (select B1 from B)
```

This query will provide us with a number denoted n, the interpretation of the result of the query will depend on the value taken by this number and on the way we have done the search i.e. the number of tuples being tested. Thus, if we have used the whole database or all the tuples of the target relation, the interpretation of the result of the query will be as in Table 2. If we have used just a sample of the database, the results of the query will have the interpretation given in the Table 3.

1. The case where the whole database is used: Let m be the number of the tuples of the target relation (m is the result of an SQL query applied to the target relation: select(*) from B).(see Table 2).

Note that the process explained in Table 2 could be more refined, especially in the case where $n \neq 0$. Indeed, before concluding hastily that the attribute of interest is not a foreign key, one can explore other tracks such as whether the attribute is optional and may be it is a secondary identifier foreign key.

2. A sample of the database is used that represents t% of the tuples of the target relation. This way of doing is very advantageous if we deal with very large databases where the number of tuples could be huge and as a consequence the time of search is also tremendous. The user has to specify the percentage (we will call it *acceptance*) above which the tested attribute is accepted as a foreign key. If the size of the sample is equal or greater than the acceptance than this case is assimilated to that where the whole instances of the database are checked.(see Table 3).

n=0	• A_1 seems to be a foreign key. This result could be refined to know the exact type of the foreign key. Indeed, we can verify whether an inclusion dependency holds from the target attribute to A_1, if yes, then we have a total foreign key else we have a standard foreign key. • The instances could change. A_1 may not be a foreign key.
$n \neq 0$ and $m-n = \varepsilon$	• A_1 is not a FK. • A_1 is a FK but there are data errors. • A_1 is a conditional FK. • A_1 is a 99% FK.
$n \neq 0$ and $m-n >> 0$	• A_1 is not a FK. • A_1 is a conditional FK. • A_1 is a loosely matching FK.
$n \neq 0$ and $m = n$	• A_1 is not a FK. • A_1 is a conditional FK. • A_1 may be an identifying FK. • A_1 may be an alternate FK.

Table 2-The interpretation of the results of the process of elicitation of foreign keys. The use of the whole database.

The most interesting case will be when the size of the sample is smaller than the acceptance. Instead of rejecting the result, we will give it a first interpretation that has to be refined later. This first interpretation will constitute a hypothesis. In fact, we will have in this case an incremental approach i.e. use another sample of the database to test the results derived from the first sample.

We can think of introducing another field in the database, a kind of counter that will be used for sampling the database. The strength of the results, sure, will depend on the size of the sample. Moreover, the results also depend of the sample itself, which can be non representative of the whole behaviour of the data. The choice of the sample has to be arbitrary. Such an incremental approach is suitable especially for large databases. Its advantage is very attractive.

Table 3 will summarise the results and their possible interpretation.

ii. Use of data analysis for discovering

The second situation to be addressed is the use of data analysis for discovering foreign keys. If we have two relations with n and m attributes respectively, to search whether some foreign keys exits between them, we have to consider different cases:

1. The case where we have no information about the primary keys

In this case we will look for inclusion dependencies. To have an idea about the difficulty of the problem let us see what happens in terms of operations i.e the number of checks of each pair of attributes to be performed. We will deal first with two relations and after that with the whole database. If we check each pair of attributes for two relations with n and m attributes where $m \leq n$, the number of operations(i.e checks) performed denoted by op will be:

n=0	• A_1 seems to be a FK (that could be conditional or standard) for x%. Test it again with (acceptance-x) x tuples, the final result will depend on this test.
n≠0 and m-n =ε	• A_1 is not a FK. • A_1 is a FK for x% but there are data errors. • A_1 is a conditional FK. • A_1 is a 99% FK. Test it again with x tuples.
n≠0 and n >>0	• A_1 is not a FK. • A_1 is a conditional FK. • A_1 is a loosely matching FK. • A_1 is a multitarget FK. Test it again with x tuples,

Table 3-results and interpretation of the recovery process of foreign keys. The use of a sample of the database.

$$op = \sum_{i=1}^{m} C_i^n C_i^m = \frac{(n+m)!}{n!m!} - 1$$

If we adopt the same approach for the whole database and check each pair of attributes occurring in the database then the number of operations (i.e checks) performed denoted by op_b will be:

$$op_b = \sum_{i=1}^{x-1} \sum_{j=i+1}^{x} \sum_{k=1}^{ni} C_k^{nj} C_k^{ni}$$

Where:

x denotes the number of relations in the database.
ni,nj denote the number of attributes of the relations i and j.
i, j and k are integers such that $1 \leq i \leq x-1$ and $2 \leq j \leq x$ and $1 \leq k \leq \max(ni)$.

As demonstrated above the number of operations to perform is tremendous. As a consequence, it seems necessary to use some heuristics to reduce the set of INDs and thus the complexity of the problem. These heuristics could be:

- Related to the question of the user: normally the user is interested in some foreign keys and not in all the foreign keys holding in a database.
- The availability of information about the domains. In that case only the attributes having compatible domains will be checked.
- The limitation of the space of search to the foreign keys composed by one or at most two attributes.
- The discard of long attributes.(example: char(200)…).

2. The case when information about keys is available

The set of INDs is normally reduced compared to the case above because we will just check the key based-INDs. In this circumstance we have to envisage two different situations for the appreciation of the complexity of the problem: the first situation is when the primary key of the target relation is simple, we will check only the simple attributes included in the "origin relation " of the foreign key. then the number of operations, op_r, will be such that

$$op_r = C_1^m = m$$

The second situation is when the primary key of the target relation is composite (with p attributes), we will check the composite attributes having p components included in the origin relation of the foreign key. Then the number of operations to perform, op_c, will be:

$$op_c = C_p^m = \frac{m!}{(m-p)!\,p!}$$

To reduce the number of operations in the above two cases, we can resort to other means. First we can think about taking the foreign key declared in the DDL as input. Secondly, To reduce the space of the search we can just consider attributes that have the same domain as one or several primary or candidate keys.

We will tackle here below two situations of discovery of foreign keys where we will be guided by the questions of the user:

- *Check if there is a foreign between relation A and relation B* knowing that B has a primary key B_1 and A has a primary key A_1 i.e. checking if the primary key of a relation is in the same time a foreign key. If we suppose A the source of the foreign key and B is its target. In this case we can have an SQL query like:

  ```
  select count (*)  from A where A. A1 not in (select B1 from B)
  ```

  The general case i.e. the attribute(s) is not necessarily the primary key of the source relation of the foreign key. The idea is to check whether some of the attributes of the relation considered as origin of foreign keys (if any) have their values included in the set of values of the primary key of the target relation. This situation will be solved by a procedural part that looks like:

  ```
  for ∀att ∈ A do
    select count(distinct att) from A
    where A.att not in (select B1 from B)
  end-for
  ```

  If we have information about the domain we can first filter the attributes of A and have as input only the attributes having the same domain as that of the primary key of the target relation, the proposed solution is:

  ```
  for ∀att ∈ A  ∧  dom(att)=dom(B1) do
    select count(distinct att)  from A
    where A.att not in (select B1 from B)
  end-for
  ```

  The interpretation of the results of the query presented above will be the same as when we have used data analysis for the purpose of proving/disproving.

- Case where the question consists in finding all the foreign keys.

  Here again, we will be just interested in standard foreign keys. We can start by discarding the optional attributes. If we have information about keys, normally, we also have

information about domains then we will reject from the remaining set of attributes those that do not have the same domain as primary keys.

As summary, we will say that the check will be performed only on the mandatory attributes having the same domain as the PKs.

We have to check each relation occuring in the database against the others. The solution for this situation will look like:

Input: The set of PKs and the names of all the relations of the database . Each primary key is coupled with the name of its origin relation.

```
for ∀ tabj∈ database do
  for ∀att ∈ tabj do
    insert into Tfk
      (select count(att) from tabj
       where tabj.att not in (select Pki  from tabi) )
  end-for
end-for
```

Where Pki denotes the primary key of the relation having the name tabi.

The interpretation of the results of the queries is the same as the discussion elaborated before.

c. Remarks

To reduce the complexity of the problem when searching for foreign keys in the whole database, we can use the results obtained from other components of the data analysis technique.We can have, for example, information from the component dealing with functional dependencies: if a FD holds in a relation and its left hand side is not a primary key. then maybe it is a FK having as origin this relation and as target another relation. It could be hints about the existence of redanduncies inside the database.

As far as we are interested by standard foreign keys, the results provided by the component searching for optional attributes could also be used.

Note that the search for foreign keys could also give hints about other constructs such as cardinalities.

B. Functional dependencies

a. Introduction

A functional dependency (FD) is a property of the semantics of the attributes and it generalises the notion of keys. It is a constraint imposed on a relation. Its formal definition is: a *functional dependency* F over schema R is a statement of the form F: X → Y where X, Y are subsets of the attributes of the relation schema R. In other words, the values of X uniquely determine the values of Y.

b. Elicitation

The use of data analysis to recover the functional dependencies could cover a certain range of situations. We will limit our selves in this work to some of them. Indeed, We can have four representative different situations, which are as follows:

- Is there a FD between A_1 and B_1 ?
- Is there a FD between $(A_1,A_2,\ldots A_n)$ and B_1?

- ❖ Generate FDs that hold in table A.
- ❖ Generate the FDs holding in the database.

In fact, we can say that, we have only three cases which match the situations 2, 3, 4. The first case will be used as an introductory example.

1. First case: it is a verification of the FD $A_1 \rightarrow B_1$ we have to check whether all the tuples with the same value in A_1 have the same value in B_1. It could be solved by an SQL query such as:

```
select count(distinct A1) as c from A t1
where 1< (select count(distinct B1) from A t2
where t1.A1=t2.A1)
```

The interpretation of the result will be based on the value taken by field c in the relation resulting from the above query.

Table 4 will summarise the discussion.

Value of the field c.	Interpretation.
c=0	• A FD holds between A_1 and B_1. The result could be different tomorrow.
c≠0 and c=ε	• No FD holds between A_1 and B_1 • A FD holds between A_1 and B_1 but the database includes erroneous data.
c≠0 and c >>	• No FD holds between A_1 and B_1

Table 4-Case1:Interpretation of the results of the recovery of FDs.

2. Second case: This situation is similar to the first one, the only difference is that the left hand side of the functional dependency is composite. In this case, we have to verify whether a functional dependency of the form A_1, A_2, $A_n \rightarrow B_1$ holds in the relation, the solution could be an SQL query as:

```
select A1, A2, ..An
from A t1
where 1<(select count(distinct B1)
         from A t2
         where t1.A1=t2.A1 and t1.A2=t2.A2... and t1.An=t2.An)
```

Table 5 will summarise the discussion.

Result	Interpretation
The resulting table is empty	• The FD holds. • The results could be different tomorrow.
The resulting table is not empty	• No FD. • A FD holds but the database includes erroneous data.

Table 5-Case2:Interpretation of the results of the recovery of FDs.

3. Third case: The number of possible FDs in a relation of degree d is 2^{2d}, it is the case when we consider that the left hand side, lhs, and the right hand side, rhs, of a FD could be simple or composite.

   That means any subset of the attributes of the relation could form the lhs, and could determine any other subset. If we discard trivial and generated FDs and those having a composite rhs, the possible number of FDs, according to [Castellanos] could be reduced to n_r where n_r is as follows:

   $$n_r = d^* (2^{d-1}-1)$$

   In order to reduce this number of possible FDs to generate for a given relation we have adopted in this work, an approach that is as follows:

   - lhs is either a simple or a composite attribute.
   - rhs of an FD is always simple attribute. The reason is that the inference rules, and especially in this case the decomposition one, allows us to reduce a list of FDs with composite rhs to a list of FDs with simple rhs. To recover the composite rhs, one can use the union inference rule.

   In this case, if we have a relation A with n attributes $A_1,\ldots A_n$, the number of the possible FDs to generate is n_{FD}. Where n_{FD} is such that:

   $$n_{FD} = \sum_{i=1}^{n-1} C_i^n (n-i)$$

   If we have information about primary keys, the FDs having these PKs as lhs are discarded which will permit us to reduce the number of possible FDs to generate.

   If an FD1 is found, all the other FDs with the lhs as superset of the lhs of FD1 are discarded. Note that these FDs are generated to permit us not checking them against the instances.

   When we have information about FKs, we can take them as lhs of a FD to be checked. The aim for doing so, is to optimise the number of operations to be performed.

   If an index is not defined on an attribute or group of attributes which is not a primary key, this attribute or group of attributes will be taken as lhs of a FD to be checked.

   For this third case, we can have a program that will be as follows:

```
Input:
   PK_set ← {PK1}
   set_checked _FDs
   set_att
   relation
Output:
   set_Fd
Begin
   set_Fd ← set_checked _FDs;
   tested ← set_checked _FDs;
   candidat ←set_att
   while candidat ≠ ∅ do
      choose x ∈ candidat
      candidat← candidat \ {x}
      if x ∉ PK_set do then
        set_couple_rhs_lhs ← Generate_rhs (att, set_att\{att});
```

```
        for ∀ (lhs,rhs) ∈ set_couple_rhs_lhs do
          if (lhs,rhs) ∉ tested then
            if check((lhs,rhs)) then
              set_Fd ← set_Fd ∪ {(lhs,rhs)};
              inference((lhs,rhs), set_att\{lhs,rhs}, tested)
            else
              candidat ← {lhs+rhs}
              /* the symbol + stands for the fact that both rhs and
                 lhs will be considered as a composed attribute
                 lhs,rhs */
            end-if
            tested ← tested ∪{(lhs,rhs)}
          end-if
        end-for
      end-if
    end-while
  end
```

PK_set: is a set that contains the primary key of the relation of interest. This set could be empty if no information is available about primary keys.

set_Checked _FDs: is a set containing the FDs that are already checked. This set could be empty.

Set_att: is a set containing all the attributes of the relation of interest. This set has to be non empty.

Relation: indicates the name of the relation of interest.

Generate_rhs: is a function that for every component of the set_att, referred as lhs, will generate all the possible rhs. The result will stored be in a structure set_couple_rhs_lhs. Note that we store the pair (lhs, rhs).

For every pair (lhs, rhs) included in the structure set_couple_rhs_lhs, we will check whether a functional dependency holds.

Check(lhs,rhs): A function that given a pair (lhs, rhs), will generate the appropriate query, execute it and return the result. Before performing all this, it will check whether this hypothetic FD was already checked or no. The script of this function could be as follows:

```
check ((lhs, rhs))
  bool result;
  string query;
begin
  query ← generate_query(lhs,rhs);
  result ← excute_query (query);
  return result.
end
```

Inference ( ): A function that given a FD that holds in the relation, will use some of the inference rules to derive new FDs. This function will add to *tested* all the inferred FDs. The parameter set_att\{lhs,rhs} represents the set of attributes to be used for the inference.

```
Inference ( (lhs,rhs), candidates, tested)
begin
  for ∀ a ∈ candidates do
    tested ← tested ∪ {(lhs+a, rhs)}
    Inference ((lhs+a,rhs), candidates\{a}, tested)
  end-for
end
```

tested: is a set containing all the pairs for which we already know whether it is an FD that holds or does not hold in the relation. This set will allow us not to test these pairs.

set_Fd: is a set containing all the FDs holding in the relation of interest.

Generate_query (lhs, rhs): will generate the appropriate query. Its pseudo-code could be as follows:

```
Select lhs
From relation t1
Where 1<(select count (distinct rhs) from relation t2 where
          t1. comp1 (lhs) = t2.comp1(lhs) and ...
          and t1.compi (lhs) = t2.compi(lhs) and ...
,         and t1. compn (lhs) = t2 compn(lhs))
```

$Comp_i$ ( ): is a function that permits to access and extract the i^{th} component of a composite lhs where $1 \leq i \leq n$. n is the number of attributes of the lhs.

Note that the transitive FDs are generated and tested, which can be seen as a weakness of this approach, but the task of trying to avoid them is arduous. Indeed, it necessitates to re-compose rhs of several attributes. Furthermore, the detection depends on the order in which FDs are recovered. By example, if we discover FDs A → B and B → C, we can avoid to check A → C, which can be derived, but if we first found A → B and A → C, B → C has to be checked anyway and A → C could not be avoided.

4. *When we deal with the whole database*

   We have to use the program used for the third case and reiterate it to take into account all the relations included in the database. Note that the interpretation of the results of the queries is the same as the third case.

5.2.2. Optional attributes

A. Introduction

Optional attributes are those attributes, which are declared using the key word NULL. It is a kind of incompleteness of the information stored in a database. NULL values modelise the missing information within a database. Null values could be assigned for different reasons [Atzeni99]:

- The value of the attribute exists but is *unknown* at the present time. An example, if we do not know the name of the course to which a student has subscribed.
- The value is *non-existent.* An example, the name of spouse when the person is not married.
- The value is non-existent or unknown, which is called *no-information*, because it tells us absolutely nothing: the value might or might not exist, and if it exists we do not know what it is.

With "NULL value" the concept of relation is extended to include the possibility that a tuple could be present by having this special value that is not included in the domain. It is a kind of extension of the domain.

Note that in relational databases, no hypothesis is made about the meaning of NULL values, but according [Atzeni99], in practice, it is always the third situation, that of the no-information value.

Note That NULL values exhibit arbitrary behaviour in SQL [Date]

- Two NULL values are considered to be duplicates of each other for the purposes of *distinct* and *unique* and *order by*.
- *Count*, *SUM* and *AVG* do not give satisfactory results in the presence of NULL values.
- If F_1 and F_2 are fields, it could happen that

  SUM $(F_1+F_2) \neq$ SUM (F_1)+SUM(F_2) in the presence of NULL values.

In this work we will not try to explore the different interpretations of NULL values, instead of that we will study special cases such as an attribute that is declared NULL but it has all its occurrences NOT NULL or an attribute that is declared NOT NULL, and in the database we find a certain value that is repeated many times, which could be a way that the programmer has chosen to model the NULL value.

B. Elicitation

We will be guided in the recovery of optional attributes, by the questions of the user.

We had envisaged several scenarios to face different situations such as:

- ❖ Is this set of attributes optional.
- ❖ When attribute A is optional what about attribute B.
- ❖ Given a set of attributes find the optional ones.
- ❖ Find the optional attributes of a given table.

Depending on the case, an appropriate SQL query will be generated. Some of the mentioned cases will be tackled in this work.

For the first question, let us say, we have a relation A with the attributes A_1, A_2, …, A_n. We want to find out which attributes are really optional i.e the attributes are declared optional but we want to verify whether they make use of this property or no. If, we call the set of attributes given by the user, set_att, we will have an SQL query such as the one presented below. This query has to be applied to all the attributes belonging to the given set of attributes.

```
for ∀Ai ∈ set_att do
  Select count(*) as C
  From A
  Where Ai is not null
end-for
```

For each attribute, we have to discuss according to the value of the field C of the resulting relation. Table 6 will summarise the results and their interpretation.

Value of C	Interpretation
C > 0 and C >>	A_i seems to be mandatory.
C > 0 and C=ε	A_i is optional.
C = 0	A_i is optional.

Table 6-Summary of the results of the elicitation of optional attributes and their interpretation.

The most interesting case is when C>0 and C>>, we can go further and try to see whether there exists some values that could give hints about the simulation of NULL values. The idea

consists on applying a query to the relation A, and visualising the attributes of interest looking for some "strange" values i.e. if for example, one of the attributes takes frequently, let us say, the value 99 or the symbol *. It could be a hint, that in fact the attribute is in fact optional.

When checking the instances, we find that all the occurrences of the attribute are not NULL. One can conclude to three different things: first, may be the database includes erroneous data, second may be the programmer is managing himself the not NULL and finally, the database has evolved.

The same situation could happen when the attribute is declared not NULL, but we find some values that occur very often (or rarely) but are meaningless. The analysis of these values, with the help of the user, will clarify the situation.

To find out the "strange" values (if any), and if we have an attribute A_i belonging to the relation A, we can have a first SQL query such as:

```
create table intermediate (Ai, C) as
( Select Ai, count (A) as C
  from A
  group by Ai)
```

On this table, we can apply another SQL query such as:

```
select * from
intermediate
order by C desc
```

The intervention of the user is required to interpret the results of the last query, and to pull out the strange values. If some values are meaningless, we could assume that it is a way to represent NULL values. This process can not be automated.

5.2.3. MIN/ MAX Cardinality

A. Introduction

To understand the notion of *cardinality*, we propose to have a quick look to the Entity Relationship (ER) model. The ER model describes data as entities[20], relationships[21] and attributes.

A *relationship type* R among n entity types[22] E_1, E_2, …, E_n defines a set of associations, or a relationship set[23], among entities from these types. A relationship type is a mathematical relation on $E_1,E_2,\ldots,E_n$, or alternatively it can be defined as a subset of the Cartesian product $E_1 \times E_2 \times \ldots \times E_n$. Each of the entity types E_1, E_2, …, E_n is said to participate in the relationship type R, and similarly each of the individual entities e_1, e_2, …, e_n is said to participate in the relationship instance $r_i = (e_1, e_2, \ldots, e_n)$ [Elmasri00].

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. The *cardinality ratio* for a (binary) relationship specifies the number of relationship instances that an entity can participate in.

The *participation constraint* specifies whether the existence of an entity depends on its being related to another entity via the relationship type.

---

[20] An *entity* is a "thing" of the real world with an independent existence [Elmasri00].

[21] When an attribute of an entity type refers to another entity type, we are in presence of a *relationship*.

[22] An *entity type* defines a collection (or set) of entities that have the same attributes. Each entity type in the database is described by its name and attributes [Elmasri00].

[23] Mathematically, a *relationship set* R is a set of relationship instances r_i, where each r_i associates n individual entities $(e_1,e_2,\ldots,e_n)$ and each entity e_j in r_i is a member of entity type E_j, $1 \le j \le n$. [Elmasri00].

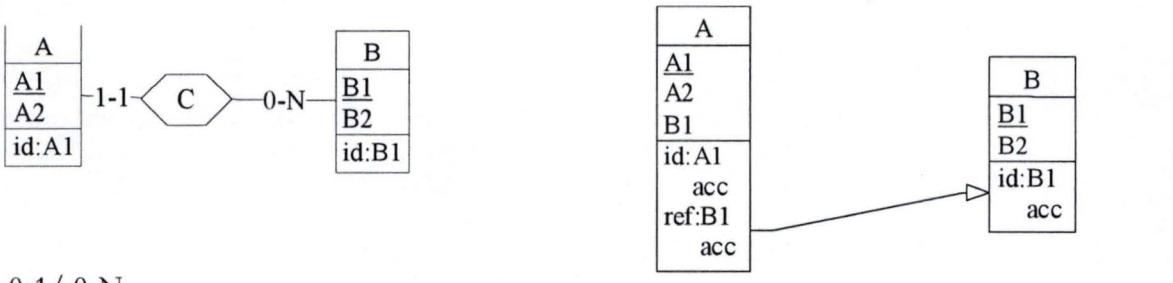
The different cardinality ratios one can have are : 0-1, 1-1, 1-N, 0-N, N-N, which will give 25 possibilities of the cardinality ratios of a relationship type. If we have two entity types A, B and a relationship type C. Then the scheme below could be read as: The entity type A participates on the instances of the relationship type C at least X times and at most Y times. the same thing could be said about B i.e. B participates to C minimum Z times and maximum T times. We denote these cardinalities by X-Y/Z-T.

Figure 8 -Entities, Relationships and Cardinalities.

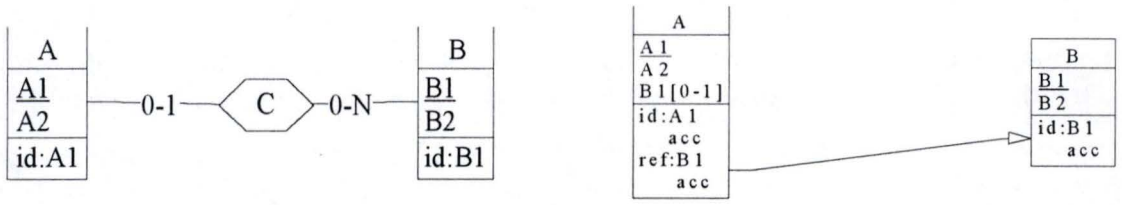
The possibilities could be reduced to 9 basic cases (if we consider that the other ones can be derived from this basic cases), which are : 0-1/0-1 , 0-1/1-1, 1-1/1-1, 0-1/0-N, 1-1/0-N, 0-1/1-N, 1-1/1-N, 0-1/N-N, 1-1/N-N. We will represent the logical schema of each case of cardinality ratio (see figure 9-1 and figure 9-2).

ER model Relational model

Case 1-1/ 0-N



Case 0-1/ 0-N



Case 1-1/ 1-N

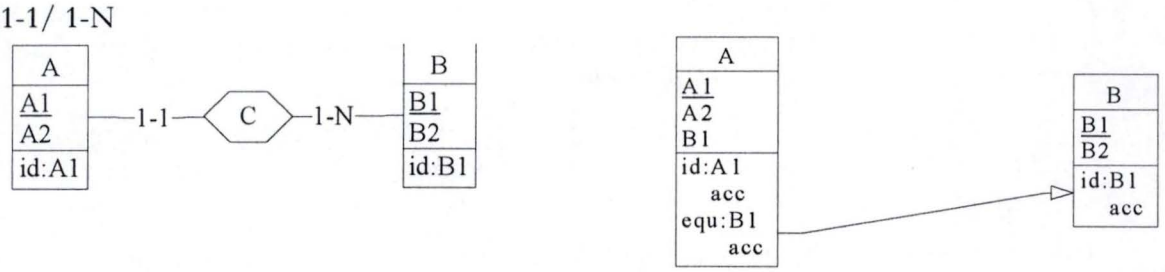
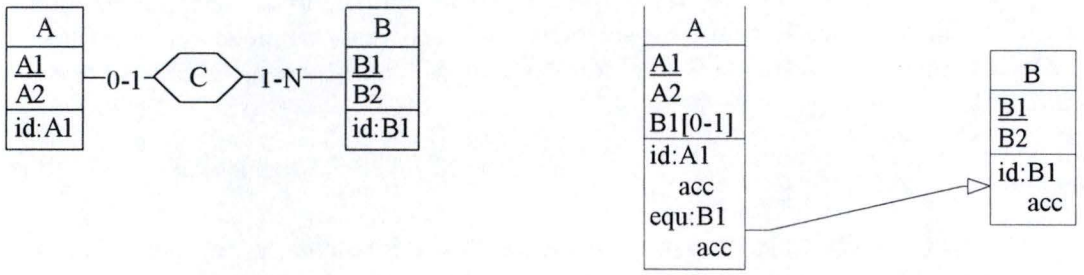


Figure 9-1-Cardinality constraints in the ER model and their correspondence in the relational model.

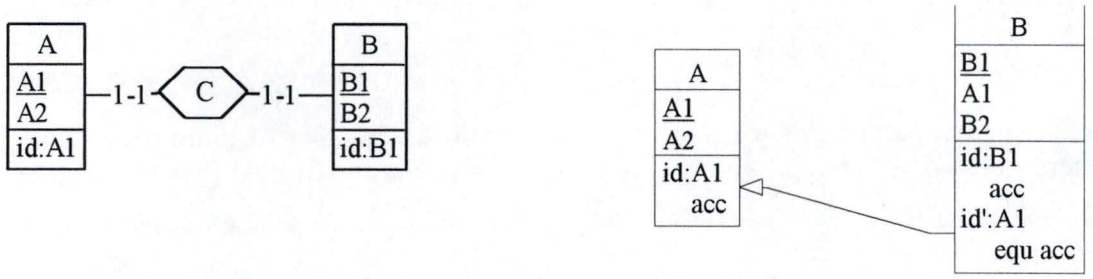
ER model

Relational model

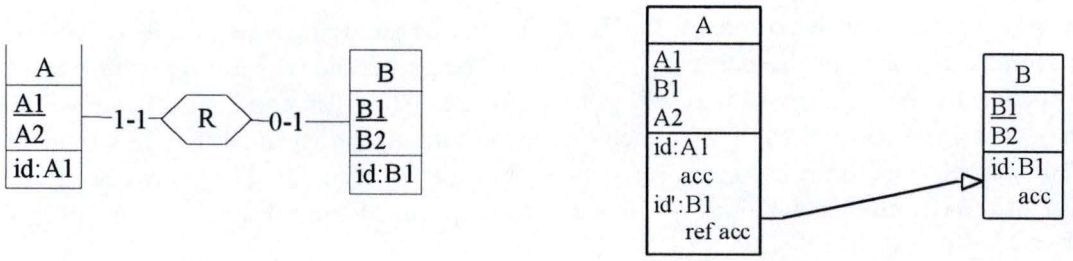
Case 0-1/ 1-N



Case 1-1/ 1-1



Case 1-1/ 0-1



Case 0-1/ 0-1

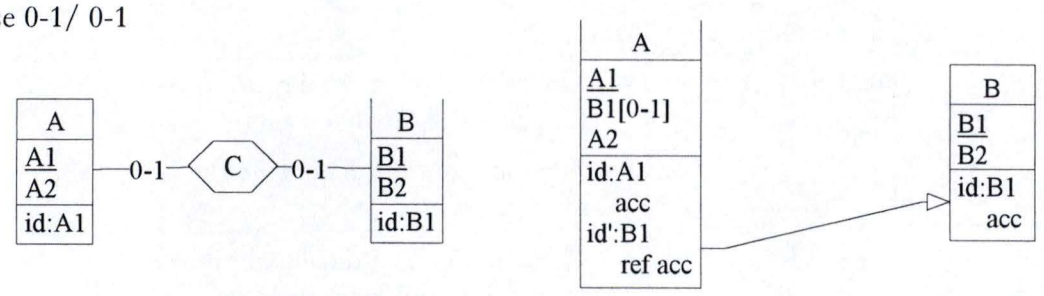


Figure 9-2--Cardinality constraints in the ER model and their correspondence in the relational model.

B. Elicitation

We will be interested in recovering the exact minimum and maximum cardinality ratios of relationship types of data models based on participation constraints. As in the relational model all what we have is a collection or a set of relation schemas, we need other information to know which relations are related. This information will sure help us to reduce the space of search.

If information about primary keys and foreign keys is available, cardinality constraint could be derived by a collection of SQL queries.

Note that, the difficulty of the recovery of the cardinalities resides on the side of the target of the foreign key. To recover the cardinalities of all the different cases of the side of the target relation, we will have an SQL query such as:

```
create table result
as ( select count(*) as c from A
group by B1
union
select 0 from B
where B1 not in ( select A. B1 from A where A. B1=B. B1))
```

While the fist part of the query will provide us with the number of times each value of B_1 is referenced, the second part, will supply count zero for the B.B1 values that are not referenced.

We will apply another query on the resulting relation to get the maximum and minimum cardinalities.

```
select min(c) as cmin, max(c) as cmax
from result
```

The interpretation of the result of the last query will be summarised in Table 7. Note that the minimum cardinality given as result of the query has to be taken as an upper bound and that the maximum has to be seen as a lower bound i.e. If, for example, the value of the field min=10 and that of max=20, we conclude that the minimum cardinality is less than or equal to 10 and the maximum cardinality is greater than or equal to 20. The recovery of the minimum and maximum cardinalities of the side of the origin of the foreign key, will also be summarised in the Table 8.

The value of c_{min}	The value of c_{max}	Interpretation
0	n>0	• The maximum cardinality, in the side of the target of the foreign key, is greater than or equal to n. • The minimum cardinality ,in the side of the target of the foreign key, is 0.
1	n>0	• The maximum cardinality, in the side of the target of the foreign key, is greater than or equal to n. • The minimum cardinality, in the side of the target of the foreign key, is less than or equal to 1.

Table 7-Max and Min cardinalities of the target relation.

State of the attribute of interest	Interpretation
Case1: the attribute referenced is an identifier of the target relation of the foreign key.	• The maximum cardinality in the side of the relation origin of the foreign key is 1.
Case2: all the components of the foreign key are mandatory.	• The minimum cardinality in the side of the relation origin of the foreign key is 1.
Case3: all the components of the foreign key are linked by a coexitence[24] constraint or are optional attributes.	• The minimum cardinality in the side of the relation origin of the foreign key is 0.

Table 8-Max and Min cardinalities of the origin relation

5.2.4. Fine grained structure of attributes:

The problem here is when a field declared as atomic is in fact a concatenation of independent fields or has an implicit decomposition. our aim is to try to recover the exact structure.

A. Elicitation

The elicitation could be guided by the questions of the user i.e. the user gives us an attribute or a set of attributes and wants to know whether an implicit decomposition is possible.

It could also be more general in the sense that what we want is to discover such attributes. In this case an SQL queries like

```
select * from tab
or
select Ai from tab
```

For the second query, Ai represents the list of the attributes already filtered out on a criteria about the length. for example we filter out all the attributes of the relation and retain only those that have, let us say, a length≥20.

While the first query would provide us with the list of all the attributes included in the relation of interest, the second one will only give the list of the attributes exceeding a certain length, note that the information about length could be derived from the DDL or system tables.

In the case that no information is available about the length we will use the first query and its result has to be analysed and a kind of filter has to be implemented which will enable us to have only the long attributes, this could be a small program that calculates the length of each attribute and to stock them elsewhere ; or we can also imagine to work on the result-relation where we will delete all the attributes having a length less than the length we have specified. Note that the minimum length is very arbitrary and it could happen that some attributes with a length smaller than the one specified are in fact a concatenation of some other small attributes.

In both cases (the first or the second query) the result-relation have to be analysed, we can imagine a program that having as input the attributes of interest will look whether separators exist or not, separators are all the characters non alphanumeric (we can remove some of these non alphanumeric characters such as ?, $...), in any way this process can not be automated 100% and the intervention of the user is required. We can also imagine another solution where an SQL query will look directly for the separators:

[24] See section dealing with existence constraint.

```
select * from tab
where ….like….
```

Example:

The application of one the ideas expressed before could provide us with a result like the one presented below:

First attribute instances

Système d'exploitation / Base de Données(MA1) / Gestion de Projet.
Pratiques de Communication / Datamining / Base de Données(MA2).

Second attribute instances

Maitrise	2	Informatique.
Maitrise	3	Informatique..
Licence	2	Sciences Economiques.
Candidature	2	Pharmacie.

Only by visualising the result and having a certain knowledge about the application domain we can derive some useful information. Note that for the second attribute of the result, one can imagine another solution. Indeed, we can spot the apparent alignment of the component of this attribute which is in this case a hint about the concatenation. Therefore, the user has to be interviewed about the semantics of such an attribute. Besides the apparent alignment, one can notice other indications. Verily, one can perceive in the structure that we have characters followed by two sequences of integers then characters. This later indication could constitute a worth hint to be exploited.

In the case of the first attribute, one can detect the presence of the separator "/". The user have to be asked about the semantics of the attribute, and only after that a conclusion could be drawn about the structure.

Note that it does not exit a general approach for the recovery of the fine grained structure of an attribute. Each case or each category has to be seen as a specific case or category. In each specific case, one has to be aware that (s) he has to show creativity to be able to perform the recovery of this implicit construct. Thus it is necessary to adopt a particular approach that we will call here *a learning by examples approach.*

5.2.5. Attribute aggregates

A compound attribute could be decomposed and represented by a sequence of apparently independent attributes. The problem here is to recover the source attribute.

A. Elicitation

Data analysis is not of much help in the recovery of attribute aggregates. Indeed the only case, where one could use data analysis when dealing with attribute aggregates, is when an existence constraint holds between the attributes. One can ask if it is not, in fact, a way to represent compound attributes. If a such constraint does not exist, we can not advance any proposal.

The recovery of attribute aggregates is a thorny issue in itself and it is made more difficult with the fact that generally the attributes do not have neither the same type nor the same length.

It will be more interesting, from our standpoint, to explore other techniques such as cliché analysis.

5.2.6. Multivalued Attributes

The aim here is to recover the multivalued attributes. This implicit construct occur when an attribute is declared single valued, and is in fact the concatenation of some values.

A. Elicitation

The process of data analysis could be guided by the questions of the user about an attribute or a set of attributes or it could be more general i.e. we explore a relation or the set of relations of the database by applying an SQL query or a set of SQL queries such as:

```
select * from tab
select att_i from tab
```

where att_i is a set of attributes specified by the user.

As an example, we can have the following as a result of one of the two queries given above:

First attribute

081-441260 * 081-725167 * 081-231626 * 00216 (0)2467586
081-441360 * 081-745167 * 081-251626 * 0033-21467586
081-441260 * 081-725167 * 081-261626 * 00212/6667586
0476-114575* 080093003

Second attribute

Anne Marie Sophie
Tintin Joseph Marc Michel

Third attribute

Rue Jules Brosteaux	n°5,	5150 Soye.
Avenue Louise	n°10,	1000 Bruxelles.
Rue Aragon	n°60,	5000 Namur.
Impasse de la Croisade	n°120,	5150 Franiere.

The result of the query has to be analysed. To be able to conclude that the attribute of interest is a multivalued attribute, the sub-attributes have to share two things. Firstly, they must have the same type, such information could be derived from the DDL. Secondly, they should have the same maximum length. Information about the length could be derived either from the DDL or by applying a program on the result relation, which will provide us with the length of the attributes.

We can then imagine, having the length of the attribute, to divide it by all its divisors, for example an attribute having a length of 40 could be divided by 2,4,5,8,10,20. For each divisor, we divide the attribute by this number, analyse the result with the help of the user and, if it is a satisfactory result, we stop the process, else we choose the next divisor. It is not a general approach and we can have erroneous results. For example, the instance " Anne Marie Sophie", if the separators are taken into account, has a length of 17, which cannot be divided by another number. The use of this variant will give good results when the fields have the same length, which is in itself a very restricting condition, especially, if we know that the declared length is an upper bound. We can also think of looking for the separators.

Note that a confusion could occur while trying to interpret the results, for example, the first attribute could be a multivalued attribute or a compound attribute, only the user could help in solving such a conflict.

5.2.7. Multiple–Domain attributes

It is when a field is used as a container for different kinds of values.

A. Elicitation

The problem can not really be addressed by data analysis. In fact, it requires knowledge of the semantics of the database. We can only deal with the special case where the components of the attribute are primary keys of other relations, which we will consider as included in the search for foreign keys.

5.2.8. Candidate keys

The aim is to recover the candidate keys of a relation when they are not declared. Note that a candidate key of a relation is one of the keys of that relation i.e. its values have to be distinct At the contrary of primary keys, it can contain Null values.

A. Elicitation

If we are interested in recovering a secondary identifier A_2 of a relation A. We have to verify that all the occurrences of A_2 are distinct. For doing so, we can have an SQL query such as:

```
select count(distinct A2) as c from A
```

The interpretation of the results provided by the above query could be summarised as follows (see Table 9):

Value of c	Interpretation
If all the fields c are equal to 1.	• A_2 seems to be a secondary identifier. The result could be different tomorrow
If at least one the fields c is greater than 1.	• A_2 is not a secondary identifier. • A_2 is a secondary identifier but there are data errors.

Table 9-Interpretation of the results of the recovery of secondary identifiers.

5.2.9. Sets behind arrays...

When a multivalued field is declared as an array or a list or a bag , it is important to recover its original structure.

A. Elicitation

We will introduce this section by an example, if we have a relation Customer with the following relation schema Customer (Id, Name, Phone1, Phone2, Phone3) where the attributes Phone1, Phone2 and Phone3 record the phone numbers of customers. One can ask whether these three attributes represent arrays, lists or bags.

We will start by extracting the attribute of interest, then we analyse its contents in the sense that we will look whether a relation exits between the different components. In other words, we apply an SQL query such as *select A_i from A*, to visualise the attribute of interest (A_i in this example), then we examine minutely the result by employing a procedural part on it. The pro-

cedural part will try to find out whether an order or /and a repetition is comprised in the structure.

unfortunately, there are several complex structures that need to be implemented in a relational database: arrays, arrays without repetitions, lists, lists without repetitions, sets, bags. And there are several ways to represent them either with multiple attributes (like in the example above), with a big attributes containing the concatenation of several values, or with a table and a FK. Analysing every possible combination of the constructs cited above deserves a complete chapter.

5.2.10. Existence constraint

One of the origins of the existence constraint is the concept of generalization / specialisation. We propose to have a quick look to the entity relationship (ER) model and especially to the concept of *generalization*. The concept of generalization represents logical links between an entity E, referred as *parent entity*, and one or more entities $E_1,\ldots, E_n$, known as *child entities*. E is more general than $E_1,\ldots, E_n$, in that it comprises them as a particular case. We say that E is a *generalization* of $E_1,\ldots, E_n$ and $E_1,\ldots, E_n$ are *specialisation* of E [Atzeni99].

The relational model does not allow the direct representation of generalization/specialisation used in the GER model [Hainaut 89]. Among the different possibilities of transformation of generalization we will retain the one that consists in collapsing the child entities into the parent entity. We will first remind the different kinds of generalization, then we will represent graphically the transformations of generalizations retained in this work.

In regard to generalizations, and following in that [Atzeni99], they can be classified on the basis of two orthogonal properties:

- A generalization is *total* if every occurrence of the parent entity is also an occurrence of one of the child entities. otherwise it is *partial*.
- A generalization is *exclusive* [25]if every occurrence of the parent entity is at most an occurrence of one of the child entities. Otherwise it is overlapping.

Apropos of the transformations of the generalizations, we can say roughly that in the case of the exclusive generalization; we will have *an exclusion constraint*, for the total generalization, we will have an *at least one constraint*; and for the overlapping generalization, no constraint is added. When a generalization is total and disjoint, we will talk about a *partition* and have an *exact-one constraint*.

The scheme below summarises (see figure 10) what was said above and it will include some other details, it consists of two parts the first one will represent the conceptual model and the second one will represent its relational equivalence. The definitions of the symbols T, P, D used in the scheme follows.

- The symbol T stands for total i.e. if A is the parent entity of B and C, we will have $B \cup C = A$ and $B \cap C \neq \varnothing$ in most cases.
- The symbol P stands for partition i.e. if A is the parent entity of B and C, we have $B \cup C = A$ and $B \cap C = \varnothing$.
- The symbol D stands for disjoint i.e. if A is the parent entity of B and C, we will have $B \cup C \subset A$ and $B \cap C = \varnothing$.

---

[25] Also known as *disjoint*.

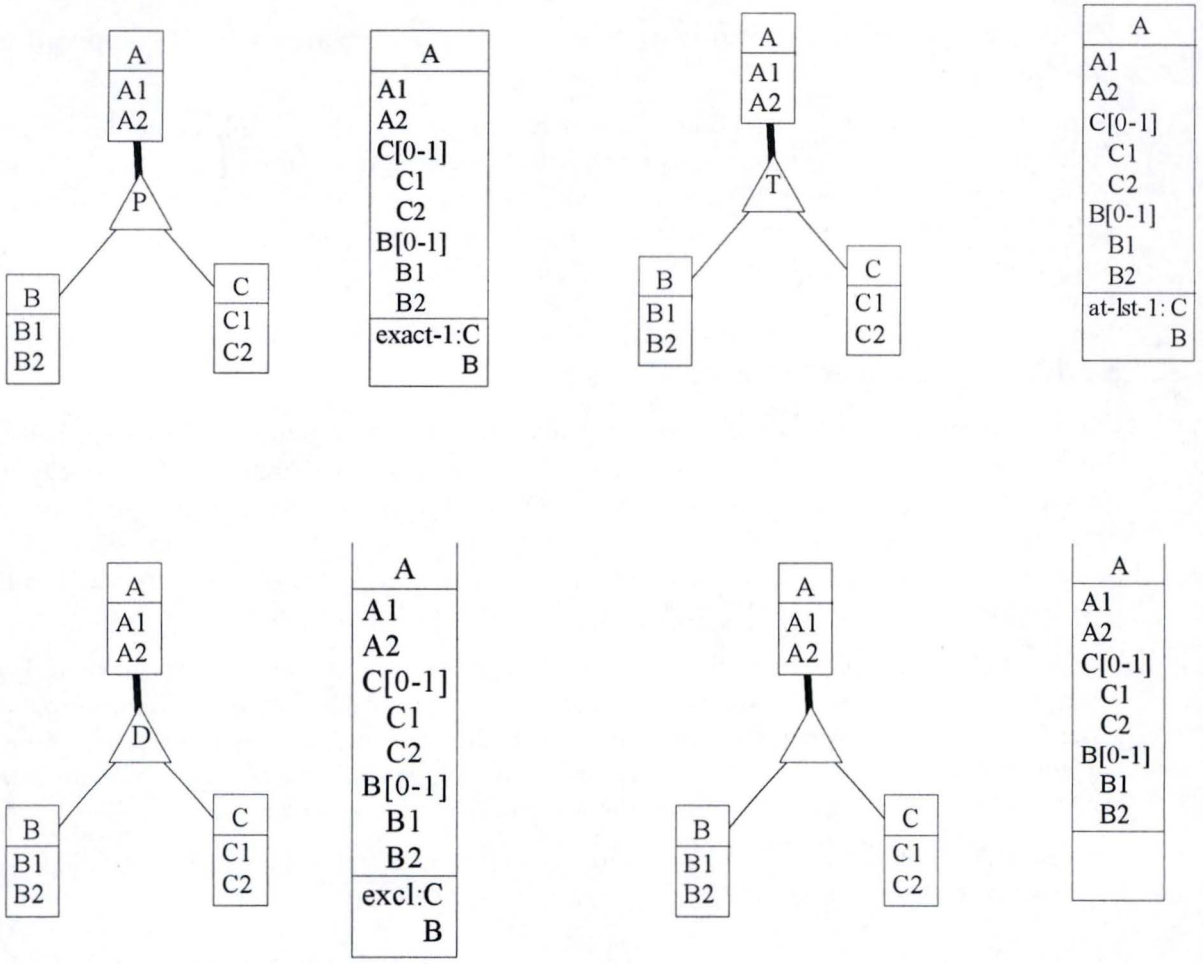


Figure 10-transformations of generalisation into the relational model.

Another source among others of existence constraints will provide us with the *coexistence constraint* which will be introduced by an example (see figure 11). If two (or more) optional attributes participate to the same coexistence constraint then they occur together or none of them is present.

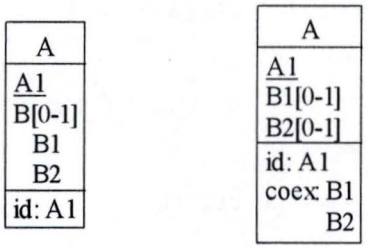


Figure 11- Disaggregation of an optional compound attribute.

Errata

Chapitre 1

Page	Endroit dans la page	Modification
p. 1-5	Phrase précédent le titre « [Navathe 87] »	Mettre « details » au singulier
p. 1-6	Dernière ligne du point « [Kalman 91] »	Insérer le mot « between » entre « relationship » et « the two relations »
p. 1-8	Phrase suivant les deux paragraphes démarqués par des puces rondes, 8e mot	Mettre « detail » au singulier
p. 1-8	Troisième case de la ligne de titre du tableau 1, 4e mot	Changer « methodMethod » en « Method »
p. 1-9	Avant-dernier paragraphe, 4e mot	Remplacer « realised » par « done »

Chapitre 2

Page	Endroit dans la page	Modification
p. 2-8	Premier ovale de la figure 5	Remplacer le mot « ext » indiçant le mot « Code », par le mot « ddl »
p. 2-8	Deuxième paragraphe commençant par « System Identification … », suivant le titre « A. Project preparation »,	Marqué le paragraphe par une puce afin de le mettre au même niveau que le paragraphe le suivant, qui commence par « Architecture recovery … »
p. 2-10	Dernier paragraphe précédent la figure 7	Monter d'un niveau le paragraphe afin de le mettre au même niveau que « Préparation » et « Basic conceptualisation », en le marquant d'une puce ronde

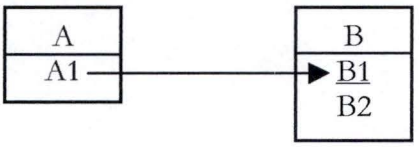
Chapitre 3

Page	Endroit dans la page	Modification
p. 3-3	Titre « 3.3.5. Multiple-doman attribute »	Mettre le dernier mot du titre au pluriel : « attributes »

Chapitre 4

Page	Endroit dans la page	Modification
p. 4-1	Dernier paragraphe de la page, 3e et dernière ligne, 4e mot	Enlever ce mot « then »
p. 4-2	Ovale « Storage Procedures » de la figure 8	Remplacer « Storage » par « Stored »
p. 4-5	5e ligne, 9e mot	Enlever ce mot « to »
p. 4-5	8e ligne du premier paragraphe, en comptant à partir de la fin, 15e mot	Mettre ce mot « customer » au pluriel
p. 4-5	2e paragraphe, 3e ligne	Ajouter « , $X \subset I$, $Y \neq \emptyset$ » à la fin de la ligne, après « $Y \subset I$ »
p. 4-6	Dernier paragraphe, dernière ligne, 2e mot	Enlever ce mot « any »
p. 4-7	Dernier paragraphe, 5e ligne, 6e mot	Mettre ce mot « V » en minuscule

Chapitre 5

Page	Endroit dans la page	Modification
p. 5-2	Première ligne, 8e mot	Enlever ce mot « will »
p. 5-3	2e paragraphe, 2e ligne	Placer le mot « and » entre « R » et « t »
p. 5-6	1er paragraphe, 5e ligne	Placer le mot « the » entre les mots « to » et « third »
p. 5-6	Note de bas de page n° 19	Changer « worsecase » en « worst case »
p. 5-13	Requête SQL du milieu de la page	Modifier la requête par « select count(*) from A where not exists (select B_1, B_2, …, B_n from B where $A.A_1=B.B_1$ and $A.A_2=B.B_2$ and …and $A.A_n=B.B_n$ »
p. 5-13	Requête SQL du milieu de la page	Ajouter la schéma suivant à la requête : 
p. 5-19	Paragraphe marqué du chiffre 2, 3e ligne	Placer entre « A_2 » et « A_n » trois points de suspension
p. 5-19	2e paragraphe, 2e ligne, 10e mot	Remplacer de mot « generated » par « degenerated »
p. 5-19	Avant-dernière ligne du programme	Enlever le mot « do »
p. 5-19	Dernière ligne du programme	Changer « (att, set_att\{att}) ; » par « (x, set_att\{x}) ; »
p. 5-20	7e ligne du programme	Changer « candidat←{lhs+rhs} » par « candidat← candidat ∪{lhs+rhs} »
p. 5-20	9e ligne du programm, 7e mot	Modifier le commentaire en changeant le mot « composed » par « composite »
p. 5-20	5e paragraphe après le programme commençant par « Generate_rhs », 2e ligne	Déplacer le mot « be » en le plaçant entre les mots « will » et « stored »
p. 5-20	Pseudocode du base de la page, après le paragraphe « Inference() : »	Précéder le pseudocode du titre : « It is pseudocode is as follows : »
p. 5-21	Requête SQL du début de la page, dernière ligne de la requête	Mettre un point, au lieu d'un espace, entre « t2 » et $comp_n$(lhs) »
p. 5-21	Paragraphe précédent le point « 4. When we deal with the whole database », avant-dernière ligne, 14e mot	Remplacer ce mot « found » par le mot « find »
p. 5-21	Dernière phrase de la page	Placer « : » après « SQL [Date] »
p. 5-22	Paragraphe précédent le pseudocode, commençant par « For this first question … », 3e ligne, 13e mot	Changer de mot « no » par le mot « not »

p. 5-26	1ère requête de la page, suivant le 3e paragraphe, dernière ligne de cette requête	Remplacer entièrement cette ligne, commençant par « where B_1 not », par « where not exists (select * from A where $A.B_1=B.B_1$ and ...)) »
p. 5-28	4e paragraphe commançant par « Only by visualising the result ... », avant-dernière et 7e ligne, 8e et 9e mots	Remplacer ces mots « two sequences » par « a sequence »
p. 5-30	3e ligne du tableau, 1ère case commençant par « If at least ... », 4e et 5e mots	Placer le mot « of » entre ces mots « one » et « the »
p. 5-24	1er paragraphe suivant la 1ère requête, 1ère ligne, les trois derniers mots	Remplacer ces trois derniers mots « a constraint exclusion » par « an exclusion constraint »
p. 5-35	3e paragraphe du titre « A. Elicitation », 1ère et 2e ligne	Mettre en indice le « i » et le « n » des « Ai » et « An », afin d'obtenir « A_i » et « A_n » respectivement
p. 5-35	Avant dernier paragraphe de la page, commençant par « Case2 could be ... », 4e ligne, 7e et 8e mots	Placer entre ces mots « Client » et « As » un point
p. 5-35	Avant dernier paragraphe de la page, commençant par « Case2 could be ... », 5e ligne, 3e et 4e mots	Placer entre ces mots « than » et « we » le mot « if »
p. 5-39	Dernier paragraphe, commençant par « Note that », après ces deux 1ers mots de la 1ère ligne	Placer « : » après ces mots « Note that »

Chapitre 7

Page	Endroit dans la page	Modification
p. 7-2	Paragraphe en dessous de la figure 12, première ligne, 14e mot	Mettre ce mot « details » au singulier
p. 7-4	Phrase précédant la figure 13, dernier mot	Mettre ce mot « details » au singulier
p. 7-7	Paragraphe du point « C. The executor », dernière ligne, 6e mot	Remplacer « dependent of » par « depends on »
p. 7-8	Dernier paragraphe, 2e et 3e phrases, mot en césure	Corriger ce mot « developmnnt » par « development »
p. 7-9	2e paragraphe, 1ère ligne, 3e mot	Corriger ce mot « extacting » par « extracting »
p. 7-11	Paragraphe « Voyager 2 » marqué d'une puce ronde, 5e sous-point marqué d'un losange sur pointe	Changer « list most assistants. » par « list of most used assistants. »

Chapitre 8

Page	Endroit dans la page	Modification
p. 8-1	1er paragraphe, avant-dernière ligne, 7e mot	Mettre la 1ère lettre de « These » en minuscule

Chapter 8

Conclusion and perspectives

This work is a contribution to the process of reverse engineering of relational databases. It is intended to recover the implicit constructs and constraints embedded in a relational database. Among the range of techniques and sources of information, *Data Analysis* is the one we have put emphasis on. Two kinds of strategies have been proposed; the first organises the reasoning when using Data Analysis, the second organises the reasoning when using the different other techniques available besides data analysis. Neither the first strategy nor the second are claimed to be general, which is an unrealistic hope for two reasons: first, reverse engineering is a complex and learning process; second, the problems tackled by reverse engineering are various, different almost each time. Instead, These strategies try to organise the reasoning and to help in solving some practical problems.

Most of the results provided by the technique of Data Analysis are approximate results, we think for example about the recovery of maximum and minimum cardinalities where Data Analysis can give only lower or higher bounds. Data analysis is used here for proving/disproving hypotheses and also for discovering them.

Although Data analysis is a costly technique, it gives information difficult to get with other techniques such as the so-called (in this work) "fuzzy" information. Therefore, we have used heuristics to reduce the space of search. Moreover, the process of recovery of some implicit constructs and constraints cannot be fully automated; the involvement of the user is required.

We can say that data analysis is fully in the spirit of the DB-Main project and is a good complement to it, but there still a lot of work to do before this work deserves its place in the DB-Main CASE tool:

- To use data analysis for other kinds of databases such as Cobol,...
- To exploit the idea expressed in this work, which consists in the use of samples of a database, especially when dealing with large databases.
- The use of samples has to be supported by a statistical study to enhance the reliability and to guarantee the quality of the results.
- When we deal with Functional dependencies, try to find a technique to evaluate the appropriate level from which we can start the search (by level we mean the number of attributes considered in the lhs of a FD).
- To see how we could couple this approach with other fields such as Data Mining.
- To analyse the possible interaction between the different components of the designed prototype.

- To proceed with the prototype design in order to cover all the implicit constructs and constraints that are encountered in relational databases.
- To allow the prototype to exchange data with other modules implementing the other recovering techniques.

A. Elicitation

The recovery of the existence constraints will be guided by the questions of the user. It will take place when these constraints are not declared or when the user, having some hypotheses, wants to verify them. For each constraint we have a range of questions that will be presented here below. We will deal only with the first kind of questions.

- For the "At least one" constraint, we will try to generate the appropriate SQL queries for the following questions:
  - ❖ Given a set of optional attributes check if a constraint at least one holds between them.
  - ❖ Given a set of optional attributes check if a constraint at least one holds between a subset of them.
  - ❖ Given a table check if a constraint at least one holds.

  For the first question, in the case we have, for example, three attributes A_1, A_2, A_3 of a relation A, we will generate the following SQL query:

  ```
  select A1, A2, A3 from A
  where not (¬ A1 ∧ ¬ A2 ∧¬ A3 Or (¬ A1 ∧ ¬ A2 ∧ A3)
  or (¬ A1 ∧ A2 ∧ ¬ A3 Or (A1 ∧ ¬ A2 ∧ ¬ A3))
  ```

  If the resulting table is empty then we can conclude that probably a constraint At least one holds between the attributes.

- For the "exact one" constraint we will generate the SQL queries suitable for the following questions:
  - ❖ Given a set of optional attributes check if an exact one constraint holds between them.
  - ❖ Given a set of optional attributes check if an exact one constraint holds between a subset of them.
  - ❖ Given a table check if an exact one constraint exits.

  For the first question, in the case we have three attributes A_1, A_2, A_3 of a relation A, we will generate the following SQL query:

  ```
  select A1, A2, A3 from A
  where not (¬ A1 ∧ A2 ∧ A3 Or (A1 ∧ ¬ A2 ∧ A3) Or (A1 ∧ A2 ∧ ¬ A3))
  ```

  If the resulting table is empty then we can conclude that probably a constraint exact one holds between the attributes.

- For the "exclusion" constraint we will generate the proper SQL queries for the following questions:
  - ❖ Given a set of optional attributes check if an exclusion constraint holds between them.
  - ❖ Given a set of optional attributes check if an exclusion constraint holds between a subset of them.
  - ❖ Given a table check if an exclusion constraint exits.

  For the first question, in the case we have three attributes A_1, A_2, A_3 of a relation A, we will generate the following SQL query:

```
select A1, A2, A3 from A
where not (¬ A1 ∧ A2 ∧ A3 Or (A1 ∧ ¬ A2 ∧ A3)
or (A1 ∧ A2 ∧ ¬ A3) Or (A1 ∧ A2 ∧ A3))
```

If the resulting table is empty then we can conclude that probably a constraint exclusion holds between the attributes.

- The "coexistence" component will generate the adapted SQL queries for the following questions:
  - ❖ Given a set of optional attributes check if a coexistence constraint holds between them.
  - ❖ Given a set of optional attributes check if a coexistence constraint holds between a subset of them.
  - ❖ Given a table check if a coexistence constraint exits.

  For the first question, in the case we have three attributes A_1, A_2, A_3 of a relation A, we will generate the following SQL query:

```
select A1, A2, A3 from A
where not (A1 ∧ A2 ∧ A3 Or ¬ A1 ∧ ¬ A2 ∧ ¬ A3 )
```

  If the resulting table is empty then we can conclude that probably a coexistence constraint holds between the attributes.

Table 10 will summarise the different situations one can face with the existence constraints:

Resulting relation	interpretation
Empty.	• An existence constraint seems to hold between the attributes. • It is an accident.
Not empty but the number of tuples is very small.	• An existence constraint could hold between the attributes but there are erroneous data • No existence constraint holds between the attributes.
Not empty but the number of tuples is very big.	• No existence constraint holds between the attributes.

Table 10-Interpretation of the results of the recovery of existence constraints.

As we can notice, the only difficulty for all the different kinds of existence constraint is to manage the number of null (not null) in the query. For example, when dealing with the at least one constraint, we need to generate a query that translates the fact, if we have n attributes, all of them are present or (n-1) or (n-2) or (n-i)...or one . Where $i \leq n-1$. A procedural part will translate this constraint (or its opposite) then we construct the SQL query, which will be the generalization of the example with three attributes.

5.2.11. Redundancies

Redundancies could be included into the database. The aim of their presence is among others to enhance performance.

We can find in [Hainaut98] two kinds of redundancies namely *Structural Redundancy* and *Normalisation Redundancy*. We will present briefly each of them.

- Structural redundancy[26]

The main problem is to detect the redundancy constraint that states the equivalence or the derivability of the redundant constructs. The expression of such constraints is of the form **C1 = f(C2, C3, ...)** where **C1** is the designation of the redundant construct. Note that expressions such as **f1(C1,C2,...)=f2(C3,C4,...)** generally do not express redundancy, but rather a pure integrity constraint, in which case no constructs can be removed

- Normalisation redundancy[27]

An unnormalised structure is detected in entity type C by the fact that the determinant of a functional dependency is not an identifier of C. Normalisation consists in splitting the entity type by segregating the components of the dependency. Note that the relationship type should be one-to-many and not one-to-one, otherwise, there would be no redundancy.(see section dealing with functional dependencies for the recovery of this kind of redundancy).

In order to normalise the schema, the recovery of the redundancies is important.

A. Elicitation

We can have different kinds of redundancy, very often each one will need an appropriate approach. This work will be based on two examples of redundancies that we will explicit below.

The structural redundancy is a composition of the two examples of redundancies that will be used in this work.

We have a relation A, its relation schema is the following A ($\underline{A_1}$, A_2, A_3, Ai...An). We can have another relation, let us call it B, such that *case1*: B($\underline{A_1}$, A_2, A_3,Ai...An) and the set of the values of B is a subset of the set of values of A or *case2*: B($\underline{A_1}$, A_i) where the set of the values of B.A_1 is equal to the set of values A.A_1.

In the real world the relation A can be a relation representing customers for example, Client (numcli, name, address, phonenum). The case1 could be a relation comprising the customers having placed an order for a value greater than 100.000Fb, let us call it, best_cli, with its relation schema best_cli (numcli, name, address, phonenum). Since best_cli has much fewer records than Client, an index on its primary key organised as a B-tree will be made of fewer levels; so an access to a record of best_cli will be faster than an access to Client.

Case2 could be a relation comprising all the customers, but just their numcli and their names for faster access than in the bigger relation. Let us call this relation, reduced_cli, its relation schema will be, reduced_cli (numcli, name). Since reduced_cli has fewer fields than Client, the records are shorter than those of Client As a consequence a big number of records of reduced_cli (more than we deal with the records of Client) can be put in a single page , reducing the number of pages to be read when accessing to all the records, and reducing the reading time.

A mixed example between case1 and case2 could be a relation comprising only the customers that have placed at least an order the last three months. It could be to keep track of the last orders, and all what we need is the identifiers of the clients and their phone numbers. Let us call this relation, last_cli, where its relation schema will be last_cli(numcli, phonenum). For

---

[26] Derived from [Hainaut98].
[27] Derived from [Hainaut98].

both case1 and case2 we have to verify if all the instances of B are included in those of A. For Case2, we have to check the opposite as well. For case1 Data Analysis is used for proving or disproving. In the case2, we have to go further and find which column is redundant, to reduce the space of search we can be helped by getting the information about the type and the length of the attributes of A and B and just compare those having the same type and length.

If we have the list of the attributes of B denoted by L, we can imagine an SQL query such as:

```
select L from B where L not in (select * from A)
```

The problem with this query is that the *in* operator does not accept a list of attributes for the second select. To compensate for that we can imagine a query such as:

```
select L from B
where L1 not in (select L1 from A) and ...
  and Ln not in (select Ln from A)
```

The problem with this query is that it will give a result that is not suitable for our case, For example, if the instances of relation A are as follows:

A_1	A_2
3	6
5	5
2	4

And the instances of relation B are as follows:

B_1	B_2
3	4
5	5
8	7
2	6

The query above ($L=(B_1,B_2)$, $L_1= B_1=A_1$, $L_2= B_2=A_2$) will give an empty table, which will lead us to wrongly conclude that relation B is included in A. The reason for that is that each value of the attribute is taken apart without considering the whole tuple.

To solve this we have to have a procedural part, which will read the relation B tuple by tuple and, for each tuple, will verify whether is it included in A or not.

Or, in the case we have information about primary keys and only in the case where B_2 is a mandatory attribute because if B_2 is optional and A_2 is optional, we will not be able to compare them, we can perform an outer join, and we will have a query like:

```
select *
from A right outer join B
where A.A1=B.B1 and (not (B.B2=A.A2)...)
```

The result for our example will be the table below:

B_1	B_2	A_2
3	4	6
2	6	4
8	7	NULL

The fact that the table is not empty indicates that B is not included in A. More generally, one can have another case (Case3) where the relation B has its own attributes besides some of the attributes of A. $B(\underline{B_1},B_2,\ldots B_m, A_1,A_j)$ and where A_1 is a foreign key to the relation A. A_j could be the name of the client for example.

5.2.12. Enumerated value domains

The aim here is to discover the set of limited and predefined values from which a given field or a set of fields is drawing its values.

A. Elicitation

We can extract the attribute of interest and analyse the values taken by this attribute. for doing so, we can have an SQL query such as:

```
select distinct A from tab.
```

Where A is the attribute or the set of attributes of interest.

This query will provide us with the values taken by the attribute for the current time (the instances of the database will give us the values taken for the moment). We mean by that we can never be sure to recover the whole set of values, we can only recover the used values. The remaining values, if any, might be recovered when applying the process in the future. If the list provided by the query is rather short, we can try to find if it is not hard coded in the application programmes.

Note that the designer could choose not to manage that himself, using SQL, for example, he can create a relation representing the domain (dom_rel) and make the attribute in the relation of interest be a foreign key to the relation dom_rel.

5.2.13. Constraints on value domains

The aim is to find out the restrictions on the allowed values of an attribute or a set of attributes.

A. Elicitation

The problem is, we can not have a general and global approach, each restriction or maybe a set of restrictions will have its proper solution. We will explore some cases.

We can look for the minimum and maximum size of an attribute, an SQL query such as:

```
select min(A),max(A) from tab
```

will provide the current min and max values, in this case the result have to be seen as a reflect of the current instances; and that the result could be different later if other instances are introduced.

If the attribute is of type date we can try to find the range.

5.3. Strategy

We will try in this section to establish a strategy to recover the different implicit constructs that could be recovered by the use of *Data Analysis* as a tool for Reverse Engineering.

Having a general method that could be able to cover all the kinds of problem is an unrealistic hope in the field of Reverse Engineering, this fact is due to the variety of problems and their specificity. As a consequence, the strategy proposed here is not claimed to be either general or mandatory. It is just an attempt to organise the reasoning and to help in lightening and reducing the complexity of the problem seen as a whole.

It is a strategy to be applied to some parts of the database i.e. the process can be incremental and as repetitive as wanted. The complexity of the Reverse Engineering projects, the difficulty of the issues addressed by this field and the quality of apprenticeship that has this process have lead us to think about a strategy to be applied to some parts of the database. Besides all these reasons, people are usually interested in some aspects of the database to be reverse engineered and not in the whole database.

In the proposed strategy, we will first of all recover the optional attributes, this decision can be warranted by the fact that this recovery will help us, in one hand, eliminating the optional attributes from the set of attributes to be checked as minimal primary keys , and as a consequence we will reduce the space of search. In the other hand, it will help us when we are dealing with cardinality constraint, for example a foreign key defined on an optional attribute is an implementation of a relationship type having minimum cardinality zero.

Then we will consider the minimal primary keys and the candidate keys, the justification is that the information about these two constructs will reduce the complexity when we will be concerned with functional dependencies.

In the third step we propose to deal with fine-grained structure of attributes, multivalued attributes, attribute aggregates and existence constraints, which can be done in any order. Different reasons have guided us to do so. Indeed, recovering for example the fine-grained structure of an attribute could provide us with hints about foreign keys and functional dependencies. Some foreign keys and functional dependencies can be missed, if the fine-grained structure of attributes was not recovered in prior of attribute aggregates. We are thinking of the case where a component of the attribute is an implicit foreign key or when a functional dependency does exist between the components. Another case worth to mention is when a functional dependency exists between one or more components of the attribute and some other attributes of the relation. In the same level, we will try to recover the multivalued attributes, which we consider as a special case of the recovery of the fine-grained structure of attributes.

The last thing in the third step will consist in the recovery of attribute aggregates and existence constraints. These two actions can be done in parallel. The recovery of attribute aggregates can give indications about functional dependencies. It is the case when a functional dependency holds among the components or when the aggregate itself is the left hand side or the right hand side of a functional dependency.

After that we will get to work on functional dependencies, which could among other things give hints or help to formulate hypotheses about the existence of foreign keys. For example, if we have a relation A ($\underline{A_1}$, B, C) and if we find a functional dependency such as B→C, we can later on try to explore the data to verify whether B is a foreign key. It could also help to elucidate redundancies. We will then grapple with foreign keys and try to find multiple-domain attributes. Finding multiple attribute structures could help in recovering the alternate foreign

keys, it is the case when the components of the field are primary keys of other relations. When all the other steps are done, we can tackle redundancies. Finally, cardinality constraints can be recovered using, in most of the cases, the results of the other actions.

The other constraints and constructs such as sets behind arrays, constraints on value domains and enumerated value domains do not seem to be strongly related to the other implicit constructs studied up to now, and can be done in any order and even in parallel with the others if wanted.

Table 11 will give a summary of the proposed strategy.

The action's number	The implicit constructs/constraints
1	Optional attributes.
2	Minimal primary keys. Candidate keys.
3	Fine-grained structure of attributes. Multivalued attributes. Attribute aggregates. Existence constraints.
4	Functional dependencies.
5	Foreign keys Multiple-domain attributes.
6	Redundancies.
7	Exact min/max cardinality
1**	Sets behind arrays...
2**	Constraints on value domains Enumerated value domains

Table 11-Summary of the proposed strategy.

Note that (**) indicates that no number is assigned to that action, and that when a step having a number from one to seven includes several sub-steps, no order in performing the sub-steps is required. The step 1** could be done in parallel with actions 1 to 7. The step 2** could be done independently of all the other actions. In the step number 3, the recovery of attribute aggregates and existence constraints can be done in parallel.

Chapter 6

The use of several techniques of elicitation

We will investigate, in this chapter, the different techniques of elicitation but the Data Analysis technique that was explored in chapter five.

Although Data Analysis is the essence of this work, we propose to have a close look at the other techniques for two major reasons. The first reason is that we believe that, in a reverse engineering project, no technique, as powerful as it could be, can pretend to cover all the problems or to be the best. Indeed, each technique, as the case may be, can bring some interesting information that we are probably unable to get with the remaining techniques. The second reason is that this work is not self containing but is intended to be integrated to the DB-main project inside which some of the techniques of recovery of implicit constructs and constraints have already been implemented. We will apply the different techniques but the documentation and the domain knowledge, which can be used for all the implicit constructs and constraints. Note that the remaining techniques will be applied to some of the implicit constructs and constraints, precisely, those that could be recovered by the Data Analysis technique. Then we propose, for some of them, a strategy organising the use of the different techniques including Data Analysis. About the strategy, we use the methodology proposed in [Hainaut98]. This methodology comprises four steps or four classes of the techniques of elicitation: *hypothesis triggering*, *hypothesis completion*, *hypothesis proving* and *hypothesis disproving*. These different classes of techniques are defined by the author as follows:

- A hypothesis triggering technique puts in light a possible implicit construct.
- A hypothesis completion technique help formulate the complete hypothesis.
- A hypothesis proving technique tends to increase the confidence in the hypothesis.
- A hypothesis disproving technique tends to decrease the confidence in the hypothesis.

6.1. Foreign key[28]

6.1.1. Elicitation Techniques

We will examine in which way the different techniques described in chapter four could help in the recovery of implicit foreign keys.

---

[28] this section is derived from [Hainaut 98]

A. Dataflow analysis

The dataflow in a program is the transport of values by variables. If a foreign key holds between a set of columns in a table and the primary key of another table (possibly the same), then, most probably, application programs will comprise some instructions establishing a flow of values between the set of columns of the first table and the primary key of the second one. If we draw a dataflow graph of the application program (nodes are variables, oriented edges represent assignments, non-oriented edges represent equality testing), we will surely find a path between the set of columns of the first table and the primary key of the second one.

B. Usage pattern analysis

If a foreign key holds between two tables A($\underline{A_1}$, A_2, A_3) and B($\underline{B_1}$, B_2, B_3) where A is referenced by B (let B_2 be that foreign key), one can observe some of the operations presented below. As an example, let A be the relation Customer (Cid, Name, Address) and B be the relation Order (O-id, O-cust, O-date), Customer and Order may be linked by the foreign key O-cust.

1. When adding a new record in Order, a check of the validity of the referencing field in Customer is performed. This operation could be formulated in an SQL-like expression as follows:

```
if exists (select * from Customer where Cid=:NC)
then insert into Order values (:NO, :NC, ...)
```

2. When removing a record in Customer, a program checks that this record is not referenced, or also removes all records from Order that reference this particular record of Customer. This operation could be formulated in an SQL-like expression as follows:

```
delete from Order
       where O-cust in (select Cid from Customer where C1);
delete from Customer where C1      (C1 stands for a condition)
```

3. Finding a record in Customer on the basis of a value of the referencing set of columns from Order. This operation could be formulated in an SQL-like expression as follows:

```
select Cid, Name, Address
from Customer, Order
where Customer.Cid = Order.O-cust and O-id=:OID.
```

4. Accessing all the records of Order for which the values of the referencing set of columns match the primary key of a record of Customer. This operation could be formulated in an SQL-like expression as follows:

```
select O-id,O-cust, O-date
from Customer, Order
where Customer.Cid=Order.O-cust and Customer.Name=:CN.
```

C. Screen, forms, report layout

When we analyse the content of screens, forms or of report layouts, it is possible to find some evidences about foreign keys.

The spatial relationships between data fields, for instance a customer name and, just below a little shifted to right, a column of order numbers, can suggest an implicit relationship.

Labels and comments included in the panels can bring information on the meaning of the various fields.

Discarded fields can bring information too. If all the columns from two tables appear in a report except one column which content is the same as the content of another one, can suggest that a foreign key holds between these two tables.

D. Program execution

Some information can be obtained by acting on an application program. For example, by trying to give an invalid customer number when entering an order, we can see if the program accepts it or not. If it is accepted then one can conclude that the customer number is not mandatory, maybe the program allows users to make an invoice to a non registered customer. If it is rejected, one can conclude that the customer number is mandatory, and maybe a strong relationship exists between the order and the customer.

E. Physical structure

- A field supported by an index, if it is not a primary key, may suggest that it is used as a foreign key, because it is common practice to wish to access foreign keys in the reverse sense.
- Data types and domains of the values of a foreign key and the referenced primary key generally match, strongly or loosely.
- A physical cluster is defined on the identifier of A and on $B.B_2$.

F. Name analysis

A simple analysis of table and column names may suggest the existence of foreign keys. we can have several situations:

- a column name looks like a shortcut of a table name, in our example O-cust.
- a column name looks like an identifying column name in another table.
- a column name contains a prefix or a suffix like "ref".
- the name of a column may suggest the role played by its table within the database.

In these examples, when we say a name looks like another one, we mean that they are the same, or maybe one is a shortcut of the other, possibly with different prefixes or suffixes i.e. they are comparable in some way.

6.1.2. Elicitation strategy

Phase	Heuristics	Short description
Hypothesis Triggering	• Name analysis • Domain knowledge • Data analysis • Screen	• Name of column $B.B_2$ suggests a table, or an external id, or includes keywords such as ref,... • Objects described by B are known to have some relation with those described by A. • Check whether a foreign key exits between A and B. • When the attributes of two different relations are included in the same screen.
Hypothesis Completion	• Name analysis • Domain knowledge	• Select table A based on the name of B_2. • Find a table describing objects which are known to have some relation with those described by B.

	• Technical constructs • Data analysis	• Search the schema for a candidate referenced table and id (with same type and length). • Search the database for a relation that has the attributes sharing the same domain and length with those comprised in B_2.
Hypothesis proving	• Technical constructs • Technical constructs • Dataflow analysis • Usage pattern • Usage pattern • Usage pattern • Usage pattern • Usage pattern • Usage pattern • Data analysis	• There is an index on B_2. • B_2 and $A.A_1$ are in the same cluster. • B_2 and $A.A_1$ are in the same dataflow graph fragment. • $A.A_1$ values are used to select B rows with same B_2 values. • B_2 values are used to select A row same A_1 value. • A.row of B is stored only if there is a matching A row. • When a row is deleted.from A. Rows with B_2 values equal to $A.A_1$ are deleted as well. • There are views based on a join with $B.B_2 = A.A_1$. • There is a view with check option selecting Bs which match A rows. • The values of B_2 are included in the values of $A.A_1$.
Hypothesis disproving	• Data analysis	• Prove that some $B.B_2$ values are not in $A.A_1$ value set

Table 12- Foreign key elicitation strategy.

6.2. Functional dependencies

6.2.1. Elicitation Techniques

The objective will be to recover the implicit functional dependencies using the different techniques described in chapter four.

A. Usage pattern analysis

We are dealing with the attributes that are not identifiers. From the point of Forward engineering, one could have several situations.

The first situation can be introduced by an example (see Table 13). We have a relation, let us call it Info, having the following relation schema, Info (id, NumProd, Prod,...), where the second attribute records the product number and the third attribute records the name of the product. The "forward engineer" is aware of the existence of a FD (NumProd→Prod), so (s)he can write a program that looks for NumProd, for example, the value 2 in the example above, and once it has found the first instance corresponding to value 2, it stops the loop and

extracts the value of the attribute Prod. It is, in the field of reverse engineering, a hint about the possible existence of a FD. Indeed, it is possible that the program does not loop because it knows that all the other values it could find are the same.

id	Numprod	Prod
1	1	butter
2	2	chocolate
3	1	butter
4	3	sugar
5	4	salt
6	2	chocolate

Table 13 - Relation info

The second situation, could be expressed by a relation, A, with its relation schema as follows: A (id, Numprod, Quantity) (see Table 14). There could exist a program that loops, when it finds a value of Numprod, it extracts the value of Quantity , then loops to extract all the possible values of Quantity (for the given value of Numprod). It is a hint that a FD does not hold between Numprod and Quantity.

id	Numprod	Quantity
1	1	20
2	1	50
3	2	200
4	1	205

Table 14-relation A.

Another situation could be detected, if we find a program such as:

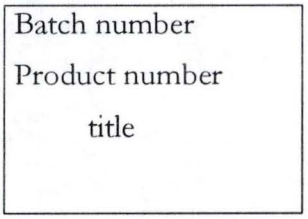
```
Accept P
Select Name from product where numpro=:P
Insert into Orderline values(:o-id, :P, :Name);
```

where P is the product number in a new orderline.

This program is performed on the relations product (numprod, Name) and Orderline (O-id, P-id, O-name). One can conclude to the probable existence of a functional dependency between numprod and name.

B. Screen, forms, report layout

We have to be aware about the positioning or the level or the arrangement of the attributes i.e. when there is a shift, it could be a hint about the possible existence of a FD between the attribute of level one (product number in the example below) and the attribute of level two (title in the example below.



C. Program execution

We can think about the case, when one enters a value in a field, another field of the same table is automatically updated.

It is the case, when we scan a bar code, we are provided with the price; the bar code and the price are stored together in an invoice table.

Another thing to be considered, is when the attribute of the first level is editable and that of the second level not i.e. when we type the first one, the second is updated.

D. Physical structure

We can think about the following case: an index is defined on an attribute which is not an identifier.

6.2.2. Elicitation strategy

We suppose that A and B are two attributes of the same relation.

Phase	Heuristics	Short description
Hypothesis triggering	• Screen • Usage pattern • Program execution • Data Analysis • Physical structure	• One field, A, which is not an identifier is editable whereas B which is next to it is not editable. • The program computes the value of B, when it sets the value of A. • When putting the value of A, the value of B appears, and it is not updateable. • Is A a rhs/lhs of a FD. • An index is defined on A, which is not a PK.
Hypothesis completion	• Screen • Program execution • Data analysis	• Have a look to the labels of the fields A and B. • When A is filled, the value of B appears. • If A is supposed to be the lhs, we try to find the rhs and conversely.
Hypothesis proving	• Usage pattern • Data analysis	• To find B, we only look for the first occurrence of A. • Check if a FD holds between A and B.
Hypothesis disproving	• Data analysis	• For a given value of A, we oftentimes find several different values of B.

Table 15-Functional dependencies elicitation strategy.

6.3. Existence constraint

6.3.1. Elicitation Techniques

A. Usage pattern analysis

If one can find a program that accesses or modifies a group of optional attributes in the same time. It could be a hint for the existence of a coexistence constraint.

Another situation could be also interesting, when we find a piece of program starting with "if" followed by a block that handles the creation or the update of a group of optional attributes in the same time, we then have a hint about a probable coexistence constraint.

If we find a program comprising a cascade of "if", each one dealing with a special case. It is a hint about a probable exclusion constraint.

B. Screen, forms, report layout

If some fields are linked to radio buttons then, it is a hint about a probable exclusion constraint.

C. Program execution

When one is obliged to fill several optional fields together on screen, it makes us think that a coexistence constraint is hidden.

When one has to choose just one field to fill, probably an exact one constraint is hidden.

When we can choose some fields among several others and we are obliged to fill at least one of them, it is probably an at least one constraint.

When we are filling several fields and we take two incompatibles ones, we have an error message. Then, we can conclude that an exclusion constraints probably holds between them.

D. Physical structure

One can verify whether a check or a trigger that verifies the existence constraint holds between some attributes.

E. Name analysis

If the attributes share the same prefix or suffix.

6.3.2. Elicitation strategy

Phase	Heuristics	Short description
Hypothesis triggering	• Usage pattern	• A and B receive values at the same time.
	• Usage pattern	• A and B never receive values in the same time.
	• Usage pattern	• When a record is created A or B, or both of them receive values.
	• Screen	• Exclusive fields are linked to radio buttons.
	• Physical structure	• A trigger verifies the existence constraint between A and B upon creation and update of records.

Hypothesis completion	• Usage pattern • Program execution • Data analysis	• Examine the grouping of the assignments in if-then-else structures. • When A and B are updated, C is also updated. • Find every C that could share the same existence constraint as A and B.
Hypothesis proving	• Usage pattern • Physical structure • Data analysis	• Examine the grouping of the assignments in if-then-else structures. • A check is defined on the relation. • Check if the constraint is satisfied for every record.
Hypothesis disproving	• Data analysis	• Find all the records for which the constraint is not satisfied.

Table 16-Existence constraint elicitation strategy.

6.4. Min/ Max Cardinality Constraint

The aim is to recover the exact minimum and maximum cardinalities.

6.4.1. Elicitation Techniques

A. Usage pattern analysis

We can have a counter, or an upper bound check, or a lower bound check, or constants.

If the access is performed directly, we can conclude that we have a maximum cardinality of 1. On the contrary, if a loop exists then we can say that the maximum is greater than one.

B. Screen, forms, report layout

We can count the number of fields to enter values of a same type.

In the case of a form to seize the personal data for new customers, if we have for example, the space for maximum five phone numbers, we can conclude that the maximum cardinality is 5. (The phone numbers are stored in a separate relation).

C. Program execution

When entering a value of an attribute, and press enter, the user is asked whether (s)he wants to enter a second value or not.

It could also be the case, if one is asked explicitly to enter from 1 to n values.

D. Physical structure

We have to verify if a trigger is present. This trigger would implement the equality constraint.

6.4.2. Elicitation strategy

Phase	Heuristics	Short description
Hypothesis triggering	• Screen	• The screen shows a defined number of fields to enter the values of A.
	• Program execution	• The program only retains the n first values entered.
	• Program execution	• The program accepts NULL values for A or not.
Hypothesis completion	• Screen	• Count the number of fields shown on the screen.
	• Usage pattern	• Constants representing the maximum and the minimum cardinality.
	• Data analysis	• Computes the minimum upper bound/ maximum lower bound of A.
Hypothesis proving	• Usage pattern	• Constants representing the maximum and the minimum cardinality.
Hypothesis disproving	• Data analysis	• To find some occurrences, the number of which is out of bound.

Table 17-Cardinality constraint elicitation strategy.

6.5. Fine–grained structure of attributes

6.5.1. Elicitation Techniques

A. Usage pattern analysis

If we find a program that extracts sub-strings from A, then we can say that A could be a concatenation of these sub-strings.

If a program performs a concatenation of strings to construct the value of A, then we can conclude that A is composed of the values being concatenated.

B. Screen, forms, report layout

The screen shows several fields, which seem to be semantically linked to A.

The length of A is of x but the length of the fields on the screen is less than x.

C. Physical structure

An attribute has a length abnormally long.

6.5.2. Elicitation strategy

Phase	Heuristics	Short description
Hypothesis triggering	• Screen	• While A is a single attribute in the database, the screen shows several fields, which seem to be semantically linked to A.
	• Data analysis	• When scrolling the content of column A on the screen, its data seem to be aligned in several sub-columns.
Hypothesis completion	• Screen	• Look at the length and the labels of the fields on the screen.
	• Usage pattern	• Extraction of sub-strings from A.
	• Usage pattern	• Concatenation of strings to construct the value of A.
	• Data analysis	• Look for separators included in A.
	• Data analysis	• Look for tabulation columns
Hypothesis proving	• Usage pattern	• Verify that an extraction of the sub-strings is performed on A.
	• Usage pattern	• Verify that a concatenation is performed to construct A.
	• Data analysis	• Verify that all the occurrences of each sub-field have the type, length...specified in the hypothesis.
Hypothesis disproving	• Data analysis	• Verify that all the occurrences of each sub-field do not have the type, length ...specified in the hypothesis.

Table 18- Fine grained structure of attributes elicitation strategy.

6.6. Optional attributes

6.6.1. Elicitation Techniques

A. Dataflow analysis

One have to verify whenever all the fields are filled or not, when filling a record.

B. Usage pattern analysis

One have to look whether all fields are set, when writing a record into the database.

C. Screen, forms, report layout

When a special symbol, such as *, appears near some fields, it is normally conceived to show that the field is mandatory.

If, while comparing two reports, one can find that certain fields are not always filled, (s)he can conclude that the corresponding attributes in the database are optional.

D. Program execution

If when filling a record, we do not fill some fields, and we get no error message from the program, then we can conclude that this field is optional. The question that remains is, how is the NULL value managed.

6.7. Field aggregates

6.7.1. Elicitation Techniques

A. Dataflow analysis

Let us say we have the following attributes: Add-num, Add-street, Add-code.

If we have a local variable, and, once we read a record, the three attributes (Add-num, Add-street and Add-code) are transferred into this local variable, and if it contains the three attributes and nothing else, then we can conclude that, in fact, these three attributes belong to the same aggregate. We can do the same comment if, while transferring the variable to the database, it includes the three attributes together.

B. Usage pattern analysis

If all the assignments of the attributes Add-num, Add-street and Add-code are grouped, we can suppose that they belong to the same aggregate.

C. Screen, forms, report layout

If some attributes are grouped together in the same panel, in a way that the group does not contain any other attribute, we can conclude that they are in fact sub-attributes of an aggregate.

D. Program execution

When the values of several attributes, for example Add-num, Add-street and Add-code, are always presented together, we can suppose that they belong to a same aggregate.

E. Name analysis

If some attributes share the same prefix or suffix, we can conclude that they may belong to the same aggregate.

6.8. Multivalued fields

6.8.1. Elicitation Techniques

A. Dataflow analysis

If the intermediate variables used to perform the transfer of the instances of the attribute of interest have one of the following structures: list, set, array... we can conclude that the attribute is probably multivalued.

B. Usage pattern analysis

If we find a loop, in which some concatenations are performed, or a slicing of an attribute into a list of sub-attributes, we can conclude that it is probably a multivalued attribute. The same conclusion could be drawn if the transfer of the instances is done in loops to/from parts of the attributes

C. Screen, forms, report layout

If the information about the length of an attribute is available, and let us say, it is of 50 characters, and we notice on the screen that we have only a field with a length of 10 characters, then we can suppose that we deal in fact with a multivalued attribute.

D. Program execution

When, for example, we are asked to enter the data one by one, and after that, the entered data appears on the same line, we can think of a multivalued attribute.

E. Physical structure

When an attribute has a length abnormally long, it might be a hint that it is a multivalued attribute.

F. Name analysis

When the name of the attribute has the plural mark, we can think that this attribute could be multivalued.

6.9. Multiple-Domain attribute

6.9.1. Elicitation Techniques

A. Dataflow analysis

The information used to fill an attribute A comes from several different sources.

B. Usage pattern analysis

If we find a program that, after reading the value of an attribute, performs a sequence of tests on its value to determine its significance, or shows that it accesses the proper relation to find out the semantics of the attribute, we can think that we are dealing with a multiple-domain attribute. For example, when we have a piece of program as follows:

```
Read A
if A. contact = 33 then customer
If A. contact = ** then supplier
```

Where A is a relation having among others an attribute contact. supplier, and customer are two other relations.

C. Screen, forms, report layout

A person is asked to enter either information 1 or information 2. For example, if one is asked to enter either his/her e-mail or his/her address, we can think that we are in presence of a multiple-domain attribute.

D. Program execution

If we have a field on screen that accepts two values of different types, we can suppose that we have a multiple-domain attribute.

6.10. Candidate keys

6.10.1. Elicitation Techniques

We will try to recover the implicit candidate keys

A. Usage pattern analysis

When we try to access the relation on basis of a field A which is not a primary key, and once the field is found, the search is stopped, it could be a hint that A is a candidate key.

B. Program execution

When creating a new record, we get an error message telling that the value already exists. We can conclude that the field could be a candidate key. Besides the uniqueness, if NULL values are accepted, we can also think about a candidate key.

C. Physical structure

If an index is defined on the attribute A. we can conclude that it is a candidate key.

6.11. Enumerated value domains

6.11.1. Elicitation Techniques

A. Usage pattern analysis

If, when a value is entered, it is checked by a comparison with a list of predefined values, we can conclude that the domain is constituted by an enumeration.

In the case that an enumerated data type is declared explicitly in the application programs and this type is used by the variables to handle the values from the database declared with a more general type (char, integer...), we can conclude to an enumerated value domain. Or, if the predefined values are stored in a local data structure in the application programs, and we no-

tice that this structure is accessed every time that the verification of the validity is performed, we can think of an enumerated value domain.

B. Screen, forms, report layout

A list of possible values from which we can choose is provided. For example, if we have a field labelled as follows: Married (Y/N). Or if we have a list with check boxes on a form. Then we can conclude that we have an enumerated value domain.

C. Program execution

We could have two different cases. The first, when we are obliged to choose the values from a list instead of entering them. The second case, will be when we are allowed to enter the values, but, if we enter a prohibited value, we get an error message. In both cases, we can think of an enumerated value domain.

D. Physical structure

If there is a check to verify the values of the attribute, then it could be an enumerated value domain

6.12. Redundancies

6.12.1. Elicitation Techniques

A. Dataflow analysis

If some fields are copied into other ones, then we can notice the presence of a redundancy. It could happen while creating or updating a field or a relation.

B. Usage pattern analysis

If exists a program or a piece of program that performs the same actions on different tables at the same time, we can conclude to the existence of a redundancy.

If it exists a trigger that, when it deals with a deletion or an insert in a relation, performs the same thing on an another relation, then we can think of a redundancy.

C. Name analysis

It can help in a very special case. When attributes in different tables have the same names or their names are comparable in a certain way, then we can conclude that a possible redundancy exists.

6.13. Constraints on value domains

6.13.1. Elicitation Techniques

A. Usage pattern analysis

If before performing a creation or an update, a verification on the validity of the different fields is done, then we can suppose that a constraint is defined on the domain.

B. Screen, forms, report layout

If we are provided with a message indicating, for example, an upper bound or a lower bound, we can suppose that a constraint is defined on the domain.

C. Program execution

If, when entering a non permitted value we get an error message, we can suppose that there exists a constraint on the domain.

D. Physical structure

If a check is defined to verify the validity of a value, then it is probable that a constraint is defined on the domain.

6.14. Sets behind arrays...

6.14.1. Elicitation Techniques

A. Usage pattern analysis

If we notice that, before inserting a new component in a structure, a test is performed to check whether the value already exists or not. we can conclude that we are dealing with one of the following, namely, a set, a list without repetitions or an array without repetitions.

When the value is accepted, we have to find out its location on the underlying structure. Is it added at the beginning, at the end or in a precise emplacement? If we are in presence of the third case i.e. the emplacement of the value within the structure is a precise one, we can conclude to the existence of an order, thus the structure cannot be a set.

Another element to explore: how is the search for the precise emplacement undertaken? If the search is done by comparisons with the other values, we can conclude to a list, a set, or a bag. An example will be, if the emplacement is done according to an alphabetic order. If the emplacement, is chosen according to an ordinal number, then we can conclude to an array. If the value is added "systematically" at the beginning or at the end of the structure, then we are in presence of either a bag or a set. If the emplacement is found out, after performing a comparison with the other values, and if besides this the test on the uniqueness of value is also done, then we have a list without repetitions. It is also possible that values are stored systematically at the end of the structure but with the adjunction of an ordinal number which denotes its real position in the structure, this can hide a list or an array.

If the "uniqueness" of the value is checked, we have a set, otherwise we have a bag.

B. Screen, forms, report layout

The following questions can help us recovering the structure:

- Do the screen layouts contain grids or lists, buttons for reordering,...?
- Are lists on reports sorted or not?

C. Program execution

Many tracks could be followed here.

- We can try to find out, after entering some values of some fields, whether a sort is done or not.
- We can look if a test on the uniqueness is done or not.
- We have to look if there exists a program that accepts holes or "doublons".
- One can look whether it is possible to enter a meaningless value that have a special specification such as "N/A" (stands for not available). This special value indicates that the emplacement will be a hole.

D. Physical structure

If we have a relation schema R (id, name, tel1, tel2, tel3), and the attribute tel1 is mandatory whereas the attributes tel2 and tel3 are optional, one can conclude that the structure is probably a set or a list rather than an array.

Chapter 7

The program architecture

The program designed in this work is intended to use the methods and heuristics proposed in chapter five, for the elicitation of the implicit constructs and constraints applicable in relational databases. It is a prototype that will use the data analysis technique to recover mainly the optional attributes, the existence and cardinality constraints, the functional dependencies and the foreign keys. In this chapter, we will present the architecture of the program designed in this work. The organisation of the different components will also be presented.

The program is made out of two main parts: the first part is written in Voyager2 which is a language designed within the DB-Main project, the second part is designed in C++. At the end of this chapter, we will present a reminder about DB-Main. The underlying reason is that the program designed in this work is intended to be part of the DB-Main project.

7.1. The Architecture

7.1.1. The General Architecture

The program designed in this work, consists of two main components namely *V2 module* and *Data Analysis module*. The relation between these two components is a *call* relation. The three other components are *Input*, *Output* files and *Communication* module. The Input part is responsible for the delivery of the information introduced by the user to the V2 module which is written in the *Voyager2* language. This input will be processed by the *V2 module* and delivered to the *Data Analysis module*. Once the information is delivered to the Data Analysis module, *Voyager2* will be suspended and freed. It is worth to note that, at this stage, the user is not obliged to wait for the results of the work (s)he has submitted to the Voyager 2 module, before using the abstract machine Voyager2 for other purposes. Then the processed input will be taken on by the data analysis module which will generate the adequate queries, access the database of interest, analyse the results of the queries and produce an output consisting of the results of the work asked on the database. These results will be communicated to the V2 module. Indeed, once the Data Analysis module has finished, it calls the communication module[29] which is responsible of calling back the V2 module.

It is important to note that the user interacts only with the V2 module both for the input and for the recovery of the results of the submitted work. As a consequence, the user is not asked to have previous knowledge about C++. In fact, the work done by the Data Analysis module is hidden to the user.

---

29 The communication module was designed by Alain Goflot who had the amiableness to provide us with this priceless part.

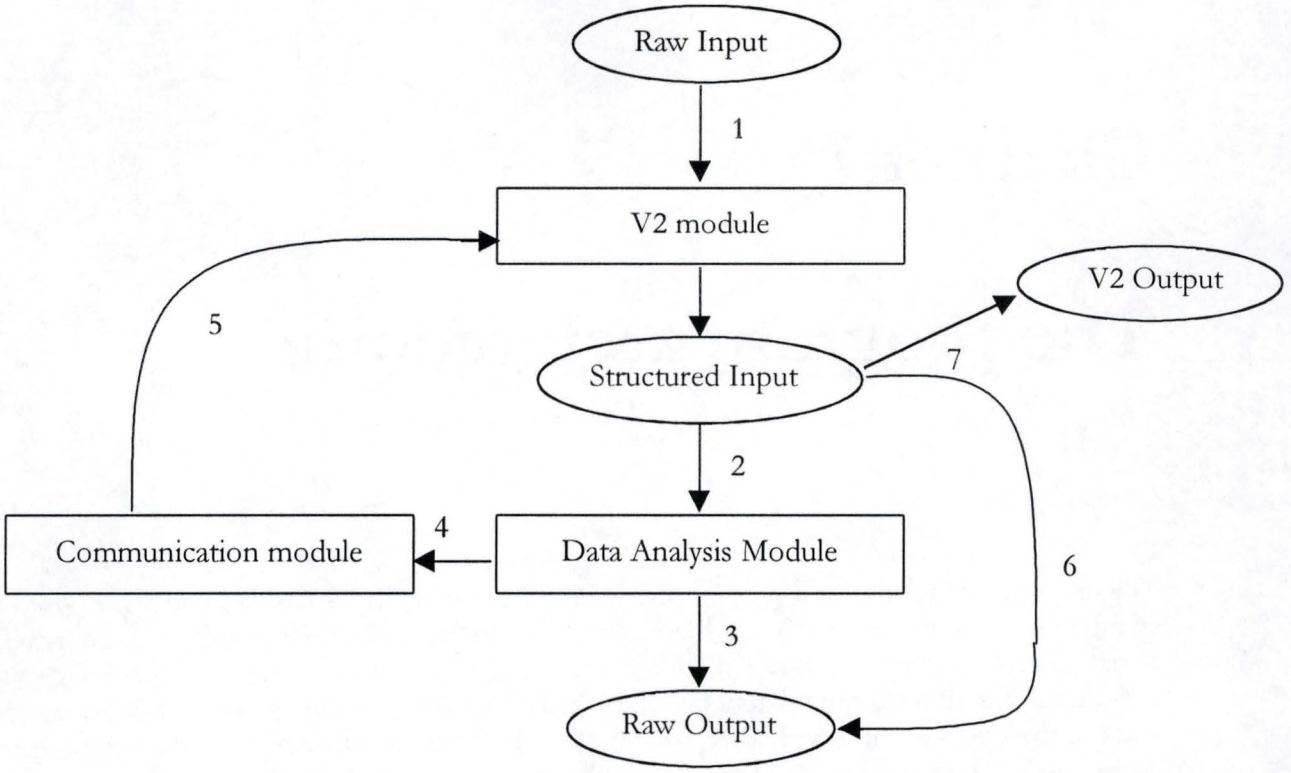


Figure 12-The General Architecture of the program.

Note that each component designed in this work, will be explained with more details in the following sections.

Figure12 gives a global view of the general architecture.

7.1.2. The Raw Input

The Input for the V2 module consists of:

- An alias of the database (to be entered by the user). We have chosen an alias instead of the name of the database to have a certain kind of genericity and to have a kind of independence of the underlying database. This alias has to be managed by the database interface module[30].
- The function of search that the user is interested in e.g. foreign key, primary key. This function has to be entered by the user.
- The name of the tables followed by their attributes. Note that the user can make his choice between entering the names manually or choosing them, in the case that the project was built using DB-Main. The idea behind the second choice is that the process of reverse engineering has already started and the user has already a schema that he wants to complete or he just wants to verify some hypotheses.
- The name of the input file: It is a file in a special format produced by the V2 module. This file is intended to be used as an input for the Data Analysis module. The information described above regarding the alias of the database, the function of search, the names of the relations and their respective attributes will be used by V2 module to fill the Input file.
- The name of the output file: The output file will be used to store the results provided by the Data Analysis module in a special format. The user has the ability to read this file, in

[30] Borland Database Engine in this work.

case, (s)he wants to follow the process and to have a more precise idea rather than to be content of the aggregated results shown in the console of Voyager2. It is the case when, for example, the user is interested by the queries used by the Data Analysis module. This file is also used by the V2 module to retrieve the results provided by the Data Analysis module.

7.1.3. The architecture of the Voyager2 module.

The first three fields introduced by the user namely, the function, the alias of the database, the names of the tables of interest and their attributes will be seized by V2 module to construct the input file. The format of this input file is as follows:

```
#Function :
#Database :
#Tables:
```

For example, if the user wants to explore a database looking for minimal primary keys, the V2 module will ask him to enter the function which will be, in this case, Pkmin. Once the function name is checked, the user will be asked to provide the alias of the database. After that, the user will be asked to specify his choice of the way to give the names of tables and their attributes i.e. to type them or to choose them. In the first case, the V2 module will check whether the names are correct or not, following in that the naming conventions used by SQL. At the end, the user is asked the input file and the output file names and paths.

When the input file is ready, the V2 module will call the Data Analysis module passing the path and names of the input and output files, and interrupt itself. This interruption is foreseen to allow the user to use Voyager 2 to do another work if needed while the Data Analysis module is working behind the scene.

An example for that, let us say that the user wants to check whether the attributes B1 and B2 belonging to table B, constitute a foreign key having as target table A with the attributes A1 and A2 as primary key. These two tables belong to a database having the alias "essai". The input file for this example will be the following:

```
#Function : FK.
#Database : essai.
#Tables: A,[A1,A2];
         B,[B1,B2].
```

The names of the input file containing the structured input and the output file together with their respective paths will be communicated to the Data Analysis module by the V2 module.

When the Data Analysis module finishes, the V2 module will access the output file, get back the results and monitor them to the user. To get back the results the V2 module has to parse the output file, and retain only the aggregated results.

7.1.4. The architecture of the Data Analysis module.

The Data Analysis module is constituted of several parts, namely the parser, the generator, the executor, the analyser and the raw output file. The generator comprises different functions or components namely the optional attributes component, the existence component, the primary key, the functional dependencies, the foreign key and the cardinality components.

The interaction between the different components is as follows:

The data structure constructed by the parser, on the basis of the information included in the input file, will be used by the generator to constitute the adequate SQL queries. The generator will call, on the basis of the field *function* of the input file, the adequate function for building the proper SQL queries. Once the SQL queries are ready, the generator passes them to the executor. Once the queries are executed on the databases, the executor passes their results to the analyser. The interpretation of the results of the queries provided by the analyser will be written in the raw output file.

The scheme on figure 13 will summarise the different components of the Data Analysis module and their interaction.

We will deal now with each component in some details.

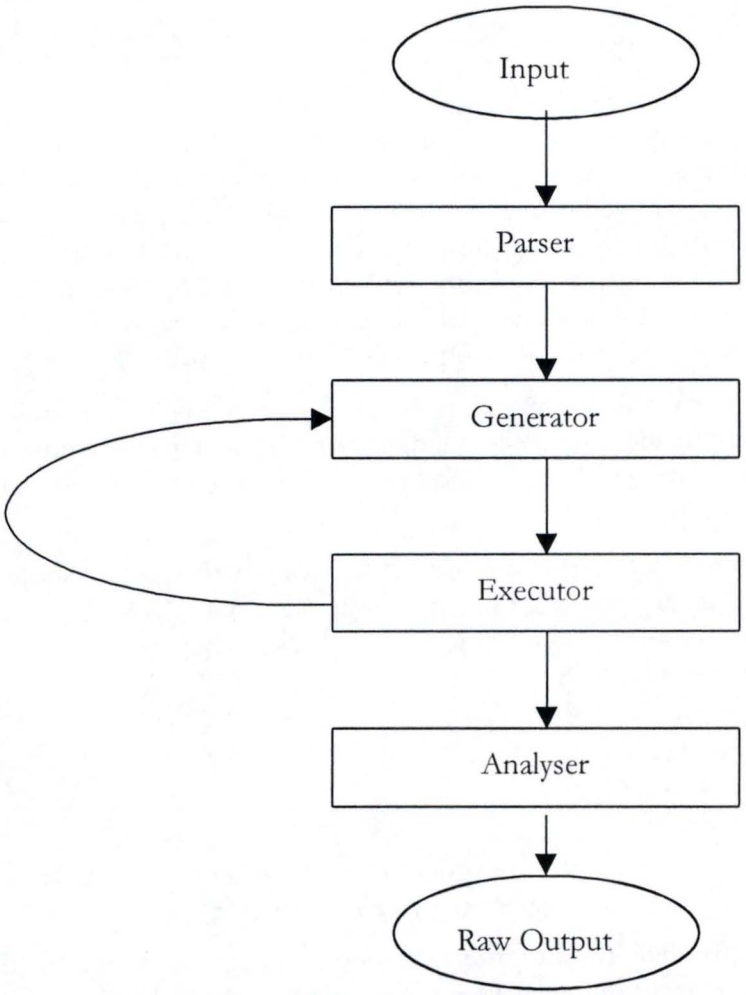


Figure 13- Dataflow graph of the data Analysis Component

A. The parser

First, the input file provided by the V2 module, will be parsed to extract the information that will be stored in a structure. A parser was designed for this purpose.

The data structure used to store the information is composed of the following elements (see the figure14).

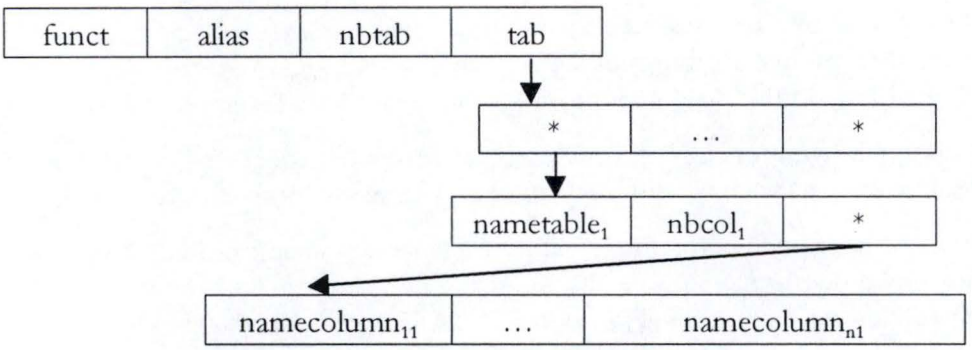


Figure 14- The data structure.

Where:

funct represents the function of search e.g. functional dependency, foreign key…
alias denotes the alias of the database of interest.
nbtab represents the number of tables on which the search will be performed.
tab is a pointer to an array of nbtab pointers to table descriptions.
nametable$_i$ designates the name of table i with $0 \leq i \leq nbtab$.
nbcol$_i$ represents the number of columns of table i of interest.
namecolumn$_{ji}$ is the name of the column number j belonging to table i, with $0 \leq j \leq nbcol_i$.

The information within this structure will be used by the generator to provide the adequate SQL queries.

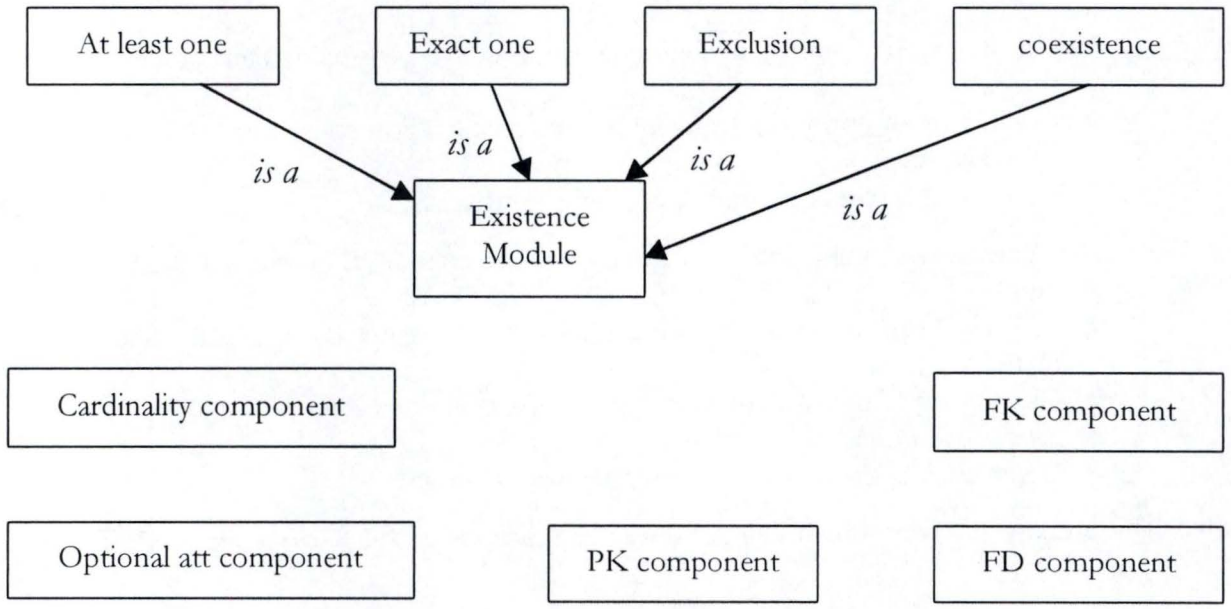


Figure 15-The components of the Generator.

B. The generator

According to the "funct" field, the generator executes one of its sub-modules (see figure 15):

- The optional attributes component has different parts depending on the question of the user. We envisaged several scenarios to face different situations such as:
  - ❖ Is this set of attributes optional?
  - ❖ When attribute A is optional what about attribute B?

- ❖ Given a set of attributes, find the optional ones.
- ❖ Find the optional attributes of a given table.
- ❖ Depending on the case, an appropriate SQL query will be generated.

- The existence component has four parts: at least one, exact one, exclusion and coexistence. In each part we have different function to answer some specific questions.

  - ♦ the "At least one" component will try to generate the appropriate SQL queries for the following questions:
    - ❖ Given a set of optional attributes, check if an at least one constraint holds between them.
    - ❖ Given a set of optional attributes, check if an at least one constraint holds between a subset of them.
    - ❖ Given a table, check if an at least one constraint holds.
  - ♦ The "exact one" component will generate the SQL queries suitable for the following questions:
    - ❖ Given a set of optional attributes, check if an exact one constraint holds between them.
    - ❖ Given a set of optional attributes, check if an exact one constraint holds between a subset of them.
    - ❖ Given a table, check if an exact one constraint exists.
  - ♦ The " exclusion" component will generate the proper SQL queries for the following questions:
    - ❖ Given a set of optional attributes, check if an exclusion constraint holds between them.
    - ❖ Given a set of optional attributes, check if an exclusion constraint holds between a subset of them.
    - ❖ Given a table, check if an exclusion constraint exists.
  - ♦ The "coexistence" component will generate the adapted SQL queries for the following questions:
    - ❖ Given a set of optional attributes, check if a coexistence constraint holds between them.
    - ❖ Given a set of optional attributes, check if a coexistence constraint holds between a subset of them.
    - ❖ Given a table, check if a coexistence constraint exists.

- The primary key component will generate the appropriate SQL queries for the following questions:

  - ❖ Given a set of attributes known to be a key, check whether they constitute a minimal primary key.

  We will explain in more detail how we deal with this question. This component uses a function that takes as arguments a function called *test* and a number specifying the level where we have to stop the search i.e. the minimum number of attributes to consider at one time, let us denote it *n*. The *test* function is responsible for calling the executor, in other words, it enables the generator to call the executor. For example, the user has entered a set composed of five attributes and wants to stop the search at level two. This function generates the inferior level by generating all the (n-1) sets of attributes, discarding each time one of the attributes. The *test* will be performed on the database to check

whether the generated sets are still keys of the relation. If the result of the test is satisfying, the procedure is called recursively.

For example, we have the attributes A, B, C, D and the minimum level is two.

In the first step, only the following combinations are generated ABC,ABD,ACD,BCD. The database is accessed to verify if these sets are keys. If the result of the test of at least one of them, is positive and the minimum level is not reached, which is the case in our example, then the inferior level will be generated and we will have AB, AC, AD, BC, BD and CD as the new sets to test.

- The functional dependencies component

  This component will generate the proper SQL queries to answer one of the following questions:

  - ❖ Is there a FD between (A1,A2,...An) and B1?
  - ❖ Generate FDs that hold in table A.
  - ❖ Generate the FDs holding in the database.

- The foreign key component

  The foreign key component will generate the adequate SQL queries for each of the following situations:

  - ❖ Case where the question consists in finding all the foreign keys.
  - ❖ Case where we have to check if a foreign key holds between relation A and relation B.

- The cardinality component

  This component is responsible of generating the proper SQL queries to recover the minimum and maximum cardinalities.

C. The executor

This component will execute the queries provided by the generator. It will access the structure given by the generator, extract each query at a time, access the database using the provided alias and create, if necessary, the intermediate tables. It could also call the generator, if the generation of a new query dependent of the results of another query.

D. The Analyser

This component is responsible for the analysis of the results of the queries provided by the excecutor. It fills the output file with some information. The output file will contain among other the result of the search done by the executor. This file has a special format that will be explained here below.

```
/***Summary***/
/***Query_p***/
/***Result_p***/
/***Interpretation ***/
/***Result_g ***/
/***Interpretation_g***/
```

Note that the three following sections could occur many times (according to the number of intermediate queries): Query_p, Result_p and Interpretation.

- The section entitled "summary" will be a reminder of the information provided by the user which is comprised in the input file, i.e. the function, the alias of the database, the tables and their attributes.
- The section entitled "query_p" is intended to comprise all the generated queries, even the intermediate ones.
- The section "result_p" contains the result of the intermediate queries.
- The section "Interpretation" contains the interpretation of partial or intermediate results.
- The section "result_g" contains the analysis of the intermediate results. The analysis will be performed on the basis of the information included in this section.
- The section "interpretation_g" contains the global interpretation. This interpretation will be delivered to the user and represents the results of the submitted work.

7.1.5. Communication module

This component is responsible for calling the V2 module when the Data Analysis module finishes.

7.2. DB-Main[31]

DB-main [Henrard 1995] is a general purpose CASE and meta-CASE environment which includes database reverse engineering and program understanding tools. Its main goal is to support all the database application engineering processes, ranging from database development to system evolution, migration and integration. In this scope, mastering DBRE is an essential requirement. The environment has been developed by the database engineering laboratory of the university of Namur, as part of the DB-MAIN project. Extensions are being developed towards federated database methodology through the InterDB project [thiran 1998] and methodological support for temporal databases (TimeStamp project). More specially, it includes the following functions, components and capabilities:

- Specifications management: access, browsing, creation, update, copy, analysis, memorising.
- Representation of the project history: processes, schemas, views, source texts, reports, generated programs and their relationships.
- A generic, wide-spectrum, representation model for conceptual, logical and physical objects: accept both entity-based and object-oriented specifications, schema objects and text lines can be selected, marked, aligned and coloured.
- Semantic and technical annotations can be attached to each specification object.
- Multiple views of the specifications (4 hypertexts and 2 graphical views); some views are particularly intended for very large schemas; both entity-based and object-oriented schemas can be represented.
- A toolbox of about thirty semantics-preserving transformational operators which provide a systematic way to carry out such activities as conceptual normalisation, or the developemnt of optimised logical and physical schemas from conceptual schemas, and conversely (i.e. reverse engineering).

[31] Derived from [Hainaut98].

- Code generators; report generators.
- Code parsers extacting physical schemas from SQL, COBOL, CODASYL, RPG and IMS source programs.
- Interactive and programmable text analysers which can be used, among others, to detect complex programming clichés in source texts, to build dataflow and dependency diagrams, and to compute program slices [Henrard 1998].
- A name processor to search a schema for name patterns and to clean, normalise, convert, or translate the names of selected objects.
- A history manager which records the engineering activities of the analyst, and which makes their further replay possible.
- Import and export of specifications.
- A series of assistants, which are expert modules in specific kinds of tasks, or in classes of problems, and which are intended to help the analyst in frequent, tedious or complex activities. It allows the analyst to develop scripts which automate frequent processes. A library of predefined scripts is provided for the most frequent activities. Six assistants are available at present: basic global transformation, advanced global transformations, schema analysis, schema integration, text analysis and reference key analysis.
- Meta CASE capabilities:
  - ❖ No tool can be claimed to solve all current and future problems. Therefore, DB-MAIN also includes a meta-development environment that allow administrators and method engineers to extend, specialise, and combine the existing concepts and functions, and to develop new ones. Extension can be performed in three ways. First, specific methods can be defined, and enforced by the tool. A method is defined by an MDL (Method Definition Language) script, compiled as a part of the repository, then enacted by the method engine.
  - ❖ Then, new processors, such as specific report and code generators, DDL analysers, or specifications checkers, can be developed in Voyager 2. This language allows to the CASE engineers (analyst or method engineers) to develop new functions which will be seamlessly incorporated in the tool without resorting to C++ programming. It is a complete 4th-generation language which offers predicative access to the repository, easy analysis and generation of external texts, definition of recursive functions and procedures, and a sophisticated list manager. It makes the rapid development of complex functions possible.
  - ❖ Finally, new properties can be dynamically added to the concepts stored in the repository. The latter is implemented as an object-base in which the tool records the current methodology, the project history, and the product specifications (mainly schemas and texts). The use of those properties is defined through Voyager 2 functions.

7.2.1. The transformation toolkit

Transformations can be used at three different levels:

- Elementary transformation (T).
- Apply transformation T to current object O. By example:

- ❖ transform the current entity type into a rel-type.
- ❖ transform the current rel-type into an object-attribute (OO-DBMS).
- ❖ desegregate the current compound attribute.
- ❖ change/add/remove prefix of names.

- Global transformation (P,T): Apply transformation T to the objects that satisfy condition P through Global Transformation Assistant actions.
- Model-driven transformation: Apply the transformations needed to make the current schema satisfy model M through Global Transformation Assistant scripts.

7.2.2. The schema analyser

The schema analyser is a tool that allows the user to look for specific constructs in a schema. It can be used in two ways:

- The first way consists in searching for constructs that verify some properties.
- The second way consists in evaluating the current schema against a specific submodel. This provides a selection of the objects that violate constraints.

Both the search and the validation criteria can be specified with predicative constraints. A complete series of such constraints are built in the DB-MAIN CASE tool.

7.2.3. The text analysers

- Extractors: analyse data structure declarations, and store their abstract representation in the repository of the tool, as a first cut logical schema: Cobol, SQL, Codasyl, IMS, RPG,...
- Pattern-matching engine: searches external texts, such as source programs, or the repository content, for instances of specific patterns.
- Dependency graph analyser: Builds the dependency graph (generalisation of dataflow diagram) of the variables of a program; nodes are variables and edges are relations defined by syntactic patterns linking variables (e.g. move A to B). Provides a visual way to trace the data flow across the variables.
- Program slicer: Given a point P in a program and one of its variables V, this processor selects all the statements that contribute to the value of V at point P. Allows the analyst to narrow the scope of the program to examine when considering how a variable is processed by this program.

7.2.4. Integration assistant

Helps merge specifications from different schemas, or within the same schema. Offers tools to integrate schemas and entity types.

7.2.5. Foreign key searching assistant

Helps the analyst in searching a complex schema for candidate foreign keys. Proposes two strategies:

Find all the candidate foreign keys referencing a given identifier, and

Find all the candidate identifiers that can be referenced by a given foreign key. A search engine allows the analyst to parameter the searching heuristics.

7.2.6. Functional extensibility

- methodological guidance: Database engineering processes can be modelled at a very fine grain. Though database design methods have been fairly standardised, developers like to follow their own way of working when they are faced with non-standard problems. This is the case for reverse engineering, but every database engineering activity can, sooner or later, require a high degree of flexibility in the way problems are solved. Though, developers still require methodological guidance, but according to their own methodology. Hence the need for powerful but flexible process models and for tools that are able to enact them. That is why a method description language (MDL) was added to the DB-MAIN CASE tool. When using an MDL method, the supporting CASE tool will explain the analyst what to do or how to do it. The CASE tool will possibly perform well defined processes that can be automated, or present to the analyst the list of processes that can or have to be performed and let him do the job in a semi-controlled way. In the meantime, all actions performed by the user are recorded in the history.

- Voyager2: The voyager 2 language allows users to develop their own DB-MAIN functions: generators, extractors and loaders, evaluators, complex transformations, etc ... These functions enrich the basic toolset without any limitation.

  Main features:

  - ❖ Communicating with the repository through either predictive or navigational queries.
  - ❖ Functions and procedures can be recursive.
  - ❖ Generic, shared, list structures are provided with powerful list operators; in particular, lists of repository objects can be built and processed; automatic garbage functions.
  - ❖ Most DB-MAIN basic tools are available from V2.
  - ❖ A V2 procedure can be dynamically added to the operators list most assistants.
  - ❖ A V2 procedure is precompiled into an internal binary code. This code is interpreted by virtual Voyager machine.
  - ❖ A V2 procedure can run external programs, either synchronously or asynchronously.

- Metaproperties: Extending the repository

  Besides the built-in static meta-properties (name, short-name, type, cardinality, SEM, TECH, etc) the user can dynamically add meta-properties to the objects of the repository. A meta-property is:

  - ❖ Typed (bool, real, char, integer, string).
  - ❖ Updateable/non-updateable.
  - ❖ Single-valued/multivalued.
  - ❖ Free or with predefined values set.
  - ❖ A meta-property can be documented.

Bibliography

[Agrawal 94] Rakesh Agrawal, Ramakrishnan Srikant, *Fast Algorithms for Mining Association Rules*, Proceedings of the 20th VLDB Conference, Santiago, Chile, 1994, pp. 487-499.

[Agrawal 93] R.Agrawal, T.Imielinski, A.Swami, *Mining Association Rules between sets of items in large databases*, Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993 pp 207-216.

[Aiken 94] Peter Aiken, Alice Muntz, Russ Richards, *Reverse Engineering Data Requirements*, Communications of the ACM, Vol.37, No 5, pp. 26-41.

[Andersson 94] Martin Andersson, *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering*, Database Laboratory, Department of Computer Science, Swiss Federal Institute of Technology, CH-1015 Lausanne, Switzerland, 1994, pp. 403-419.

[Andersson 96] Roland Martin Andersson, *Reverse Engineering of Legacy Systems: from value-based to Object-Based Models*, Thèse n°1521 (1996), Lausanne 1996.

[Andersson 98] Martin Andersson, *Searching for semantics in COBOL legacy applications*, Data Mining and Reverse Engineering, S. Spaccapietra & F. Maryanski (Eds), 1998, pp. 162-183.

[Atzeni 99] Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Riccardo Torlone, *Database Systems Concepts, languages and architectures*, McGraw-Hill, United Kingdom, 1999.

[Batini 86] C. Batini, M. Lenzerini, S.B. Navathe, *A Comparative Analysis of Methodologies for Database Schema Integration*, ACM Comparing Surveys, Vol. 18, No. 4, December 1986, pp. 324-364.

[Beeri 87] Catriel Beeri, Michael Kifer, *A Theory of Intersection Anomalies in Relational Database Schemes*, Journal of the Association for Computing Machinery, Vol. 34, No. 3, July 1987, pp. 544-577.

[Bell 93] D.A.Bell, *From Data Properties to Evidence*, IEEE transactions on knowledge and data engineering, vol. 5, no. 6, 1993, pp. 965-972.

[Blaha 97] Michael R. Blaha, *Dimensions of Database Reverse Engineering*, Proceedings of the 4th Working Conference on Reverse Engineering (WCRE'97), October 6-8, Amsterdam, Netherlands, IEEE Computing Society, 1997; pp. 176- 183.

[Biggerstaff 93] Ted J. Biggerstaff, *The Concept Assignment Problem in Program Understanding*, IEEE, 1993, pp. 482-498.

[Briand 88] H. Briand, C. Ducateau, Y. Hebrail, D. Herin-Aime, J. Kouloumdjian, *From minimal cover to entity-relationship diagram*, Entity-Relationship Approach, S.T. March (Editor) Elsevier Science Publishers B.V. (North-Holland), ERI, 1988, pp. 287-304.

[Cai 91] Yandong Cai, Nick Cercone, Jiawei Han, *Attribute-Orientated Induction in Relational Databases*, Proceedings of AAAI Workshop on Knowledge discovery in Databases, 1991, pp. 213-228.

[Casanova 93] Marco A.Casanova, Luiz Tucherman, Alberto H.F. Laender, *On the design and maintenance of optimized relational representations of entity-relationship schemas*, Data & Knowledge Engineering 11, North-Holland, 1993, pp 1-20.

[Castellanos 93] Malu Castellanos, *A Methodology for Semantically Enriching Interoperable Databases*, in Proceedings of the 11th British National Conf. On Databases, 1993, pp.58-75.

[Castellanos] Malu Castellanos, Felix Saltor, *Extraction of Data Dependencies*, in Report LSI-93-2-R, University of Catalonia, Barcelona, 1993.

[Chiang 94] Roger H.L.Chiang, Terence M.Barron, Veda C.Storey, *Reverse engineering of relational databases : Extraction of an EER model from a relational database*, Data & Knowledge Engineering 12, 1994, pp 107-142.

[Chiang 97] Roger H.L. Chiang, Terence M. Barron, Veda C. Storey, *A framework for the design and evaluation of reverse engineering methods for relational databases*, Data & Knowledge Engineering 21, 1997, pp. 57-77.

[Chikofsky 90] Elliot J. Chikofsky, James H. Cross II, *Reverse Engineering and Design Recovery:A Taxinomy*, IEEE Software, vol. 7(1), 1990, pp. 13-17.

[Cosmadikis 90] Stavros S. Cosmadakis, Paris C. Kanellakis, Moshe Y. Vardi, *Polynomial-Time Implication Problems for Unary Inclusion Dependencies*, Journal of the Association for Computing Machinery, Vol. 37, No.1, January 1990, pp. 15-46.

[Date] Date, C. J., *Relational database writings: 1991-1994*, Addison-Wesley Reading (Mass.), 1995.

[Date 92] C.J.Date, Ronald Fagin, *Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases*, ACM Transactions on Database Systems, Vol.17, No.3, September 1992, pp. 465-476.

[Davis 88] Kathi Hogshead Davis, Adarsh K. Arora, *Converting a relational database model into an entity-relationship model*, 17th Int. Conf. on Entity-Relationship Approach (ER'98), 1998, pp. 271-285.

[Debaud 96] J-M. Debaud, *Lessons from a Domain-based Reengineering Effort*, Proceedings of the third Working Conference on Reverse Engineering (WCRE'96), 1996.

[Delmal 98] Pierre Delmal, *SQL 2: Application à Oracle, Access et RDB*. De Boek & Larcier, Belgium 1998, Second edition.

[Elmasri 00] Ramez Elmasri, Shamkant B. Navathe, *Fundamentals of database Systems*, Addison-Wesley, 2000 third edition.

[Englebert 99] V. Englebert, Voyager 2 (version 5.0) - Reference manual, DB-MAIN technical manual, December 1999, public. Institut d'informatique, FUNDP

[Englebert 98] Englebert, V., Voyager 2 (version 4.0) - Reference manual, DB-MAIN technical manual, November 1998, public. Institut d'informatique, FUNDP

[Englebert 96] Englebert V., Henrard J., Hick J.-M., Roland D., Hainaut J.-L., DB-MAIN: un atelier d'ingénierie de base de données, Ingénierie des Systèmes d'Information 4(1), 1996, HERMES-AFCET.

[Fahrner 95] Christian Fahrner, Gottfried Vossen, *A survey of database design transformations based on the Entity-Relationship model*, Data & Knowledge Engineering, vol. 15, 1995, pp. 213-250.

[Fonkam 92] M.M. Fonkam, W.A. Gray, *An Approach to Eliciting the Semantics of Relational Databases*, Proceedings of the 4th Int. Conf. CaiSE'92, Manchester, UK, May 1992, Springer-Verlag LNCS 593, pp. 463-480.

[Frazer 92] Frazer A., *Reverse engineering – hype, hope or here ?, Software Reuse and Reverse Engineering in Pratice*, Applied Information Technology, The Institute of Software Engineering, edited by P.A.V. Hall, Chapman & Hall, United Kingdom, vol. 12, 1992.

[Gottlob 87] Georg Gottlob, *Computing Covers for Embedded Functional Dependencies*, Proceedings of the Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, March 23-25, 1987, San Diego, California, 1987, pp. 58-68.

[Hainaut 93] J-L. Hainaut, C. Tonneau, M. Joris, M. Chandelon, *Schema Transformation Techniques for Database Reverse Engineering*, in Proc. of the 12th Conference on Entity-Relationship Approach, Arlington-Dallas, December. 1993.

[Hainaut 93a] Hainaut, J.-L., Chandelon, M., Tonneau, C., Joris, M., *Contribution to a Theory of Database Reverse Engineering*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, May 1993, IEEE Computer Society Press.

[Hainaut 95a] Hainaut, J-L., Englebert, V., Henrard, J., Hick, J-M., Roland, D., *Requirements for Information System Reverse Engineering Support*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Toronto, July 1995, IEEE Computer Society Press.

[Hainaut 95b] Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.-M., Roland, D., *DB-MAIN: a Database Reverse Engineering CASE tool*, in Proc. of the 6th European Conference on Next Generation CASE tools, Jyvaskyla, Finland, July 1995.

[Hainaut 96a] Hainaut J.-L., Englebert V., Henrard J., Hick J.-M., Roland D., *Database Reverse Engineering : from Requirement to CARE tools*, Journal of Automated Software Engineering, 3(2), 1996, Kluwer Academic Press.

[Hainaut 96b] Hainaut J.-L., Roland D., Englebert V., Hick J.-M., Henrard J., *Database Reverse Engineering - A Case Study*, in Actes du 2ème Forum International d'Informatique Appliquée, Tunis, 12-14 mars 1996.

[Hainaut 96c] Hainaut J.-L., Henrard J., Roland D., Englebert V., Hick J.-M., *Structure Elicitation in Database Reverse Engineering*, in Proc. of the 3rd Working Conf. on Reverse Engineering, IEEE Computer Society Press,November 1996.

[Hainaut 97] J-L. Hainaut, J-M. Hick, J. Henrard, V. Englebert, D. Roland, *The Concept of Foreign key in Reverse Engineering, A Pragmatic Interpretative Taxonomy*, Technical report, University of Namur, Computer Science Department, March 1997.

[Hainaut 98] J-L. Hainaut, *cours de base de données- matière approfondie*, University of Namur, 1998.

[Han 92] Jiawei Han, Yandong Cai, Nick Cercone, *Knowledge Discovery in Databases : An Attribute-Orientated Approach*, Proceedings of the 18th VLDB Conference, Vancouver, British Columbia, Canada, 1992, pp. 547-559.

[Han 96] Jiawei Han, Yongjian Fu, Wei Wang, Jenny Chiang, Wan Gong, Krzystof Koperski, Deyi Li, Yijun Lu, Amynmohamed Rajan, Nebojsa Stefanovic, Betty Xia, Osmar R.

Zaiane, DBMiner: *A system for Mining Knowledge in Large Relational Databases*, KDD, 1996, pp. 250-255.

[Henrard 95] Henrard, J., Englebert, V., Hick, J.-M., Roland, D., Hainaut, J.-L., *DB-MAIN: un atelier d'ingénierie de bases de données*, in Proc. of the "11emes journées Base de Données Avancées", Nancy (France),September 1995.

[Henrard 98] Henrard, J., Englebert, V., Hick, J-M., Roland, D., Hainaut, J-L., *Program understanding in databases reverse engineering*, in Proceedings of DEXA'98, Vienna, August 1998.

[Henrard99] Henrard, J. , Hainaut, J.-L., Hick, J.-M., Roland, D., Englebert, V., *Data structure extraction in database reverse engineering*, REIS'99 Workshop, Paris, November 1999.

[Johannesson 93] Paul Johannessen, *Using Conceptual Graph Theory to Support Schema Integration*, 13th Int. Conf. On Entity-Relationship Approach (ER'93), 1993, pp. 283-296.

[Johannesson89] Paul Johannesson, Katalin Kalman, *A Method for Translating Relational Schemas into Conceptual Schemas*, in Entity-Relationship Approach to Database Design and Querying, Proceedings of the Eight International Conference on Enity-Relationship Approach, Toronto, Canada, 18-20 October, 1989, North-Holland, 1989.

[Johannesson 94] Paul Johannesson, *A Method for Transforming Relational Schemas into Conceptual Schemas*, in Proceedings of the 10th International Conference on Data Engineering, February 14-18, 1994, Houston, Texas, USA, IEEE Computer Society, 1994, pp. 190-201.

[Kalman] Katalin Kalman, *Implementation and Critique of an algotithm which maps a Relational Database to a Conceptual Model*, in Proc. 3rd Int. Conf. On Advanced Information Systems Engineering, Springer Verlag, LNCS 498, 1991, pp. 393-415.

[Kryszkiewicz] Marzena Kryszkiewicz, *Representative association rules and Minimum condition maximum Consequence Association Rules*, Principles of Data Mining and knowledge discovery, 1999, pp361-369.

[Leuchner 88] J. Leuchner, L. Miller, G. Slutzki, *A Polynomial Time Algorithm for Testing Implications of a Join Dependency and Embodied Functional Dependencies*, in Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988, pp. 218-224.

[Levene 99] Mark Levene, George Loizou, *A guided tour of Relational Databases and Beyond*, Springer, London 1999.

[Li 93] Liwu Li, *Fast In-Place Verification of Data Dependencies*, IEEE Transactions on knowledge and data engineering, vol. 5, no. 2, 1993, pp. 266-281.

[Mannila 94] Heikki Mannila, Kari-Jouko Räiha, *Algorithms for inferring functional dependencies from relations*, Data-knowledge Engineering 12, 1994, pp. 83-99.

[Markowitz 90] Victor M. Markowitz, Johann A. Makowsky, *Identifying Extended Entity-Relationship Object Structures in Relational Schemas*, IEEE Transactions on software engineering, Vol. 16, no. 8, 1990, pp. 777-790.

[Mc Kearney 96] Stephen Mc Kearney, Huw Roberts, *Reverse Engineering Databases for Knowledge Discovery*, KDD, 1996, pp 375-378.

[McGill 92] McGill R., *Reverse engineering – not yet ?*, Software Reuse and Reverse Engineering in Pratice, Applied Information Technology, The Institute of Software Engineering, edited by P.A.V. Hall, Chapman & Hall, United Kingdom, vol. 12, 1992.

[Mfourga 97] N. Mfourga, *Extracting Entity-Relationship Schemas from Relational Databases : A Form-Driven Approach*, in Proceedings 4th Working Conf. On Reverse Engineering, October 6-8, 1997, Amsterdam, Netherlands, pp. 184-193.

[Narasimhan 93] Badri Narasimhan, Shamkant B. Navathe, Sundaresan Jayaraman 1993, *On mapping Er and relational models into 00 Schemas*, 12th Int. Conf. on Entity-Relationship Approach (ER'93), 1993, pp.402-413.

[Navathe 87] S.B. Navathe, *Abstracting relational and hierarchical data with a semantic data model*, Proceedings of the 6th international conference on Entity-Relationship Approach (ER'87), 1987, pp. 305-333.

[Premerlani 94] William J. Premerlani and Michael R. Blaha, *An Approach for Reverse Engineering of Relational Databases*, Communications of the ACM, vol. 37, No. 5, 1994, pp. 42-49.

[Petit 96] Jean-Marc Petit, *Fondement pour un processus réaliste de rétro-conception de bases de données relationelles*, thèse de doctorat de l'université Claude Bernard, Lyon I, 1996.

[Roddick 93] John F. Roddick, Noel G. Craske, Thomas J. Richards, *A Taxonomy for Schema Versioning Based on the Relational and Entity Relationship Models*, ER 93, pp. 137-148.

[Rosenthal 94] Arnon Rosenthal, David Reiner, *Tools and Transformations-Rigorous and Otherswise-for Practical Database Design*, ACM Transactions on Database Systems, Vol.19, No.2, June 1994, pp.167-211.

[Shao 96] J. Shao *Using Rough Sets for Rough Classification*, 7th int. Workshop on Databases and expert systems Applications, 1996, pp. 268-357.

[Signore 94] Oreste Signore, Mario Loffredo, Mauro Gregori, Marco Cima, *Reconstruction of ER Schema from Database Applications : a Cognitive Approach*, 13th int. Conf. On Entity-Relationship Approach Springer Verlag, Springer Verlag, LNCS 881, 1994, pp. 387-402.

[Storey 91] Vada C. Storey, *Relational database design based on the Entity-Relationship model*, Data & Knowledge Engineering, Vol. 7, North Holland, 1991, pp. 47-83.

[Tan 97] Hee Beng Kuan Tan, Tok Wang Ling, *A method for the recovery of inclusion dependencies from data-intensive business programs*, Information and Software Technology 39, 1997, pp. 27-34.

[Tari 98] Zahir Tari, Omran Bukhres, John Stokes, Slimane Hammoudi, *The Reengineering of Relational Databases based on Key and Data Correlations*, Data Mining and Reverse Engineering S. Spaccapietra & F. Maryanski (Eds), 1998, pp. 184-215.

[Thiran98] Thiran, Ph., Hainaut, J-L., Bodart, S., Deflorenne, A., Hick, J-M., *Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE Support: the InterDB Approach*, in Proceedings of Coopis'98, New-York, August 1998, IEEE Computer Society Press, 1998.

[TIM 97] http://www.csi.uottawa.ca/~tcl/kbre/overview.html

[Tseng 93] Frank S.C. Tseng, Arbee L.P. Chen, Wei-Pang Yang, *Refining imprecise data by integrity constraints*, Data & Knowledge Engineering, Vol. 11, 1993, pp. 299-316.

[Van der Lans 00] Rick F. van der Lans, *Introduction to SQL: Mastering the Relational Database Language*, Addison-Wesley, Great Britain 2000, third edition.

[Weddell 92] Grant E. Weddell, *Reasoning about Functional Dependencies Generalized for Semantic Data Models*, ACM Transactions on Database Systems,Vol.17, No. 1, March 1992, pp. 32-64.

[Weiser 84] Mark Weiser, *Program Slicing*, IEEE Transactions on software engineering, vol. SE-10, IEEE, 1984, pp. 352-357.

[Weiss 98] Sholom M. Weiss, Indurkhya Nitin, *Predicative Data Mining: a practical guide*, Morgan Keufmann, USA 1998.

[Yen 97] Show-Jane Yen, Arbee L.P. Chen, *An Efficient Data Mining Technique for Discovering Interesting Association Rules*, in Proceedings of the Eighth International Workshop on Database and Expert Systems Applications, DEXA '97, September 1-2, 1997, Toulouse, France, IEEE Computer Society Press, 1997, pp. 664-669.

[Yoon 93] Jong P. Yoon, Larry Kerschberg, *A Framework for Knowledge Discovery and Evolution in Databases*, IEEE transactions on knowledge and data engineering, vol.5, no.6, 1993, pp. 973-984.