Journal. of Automated Software Engineering, 3(1), Kluwer Acad. Pub.

Database Reverse Engineering : from Requirements to CARE tools[1]

J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
Institut d'Informatique, University of Namur, rue Grandgagnage, 21 - B-5000 Namur
jlh@info.fundp.ac.be

Abstract. This paper analyzes the requirements that CASE tools should meet for effective database reverse engineering (DBRE), and proposes a general architecture for data-centered applications reverse engineering CASE environments. First, the paper describes a generic DBMS-independent DBRE methodology, then it analyzes the main characteristics of DBRE activities in order to collect a set of desirable requirements. Finally, it describes DB-MAIN, an operational CASE tool developed according to these requirements. The main features of this tool that are described in this paper are its unique generic specification model, its repository, its transformation toolkit, its user interface, the text processors, the assistants, the methodological control and its functional extensibility. Finally, the paper describes five real-world projects in which the methodology and the CASE tool were applied.

Keywords: reverse engineering, database engineering, program understanding, methodology, CASE tools

1. Introduction

1.1. The Problem and its Context

Reverse engineering a piece of software consists, among others, in recovering or reconstructing its functional and technical specifications, starting mainly from the source text of the programs (IEEE, 90) (Hall, 92) (Wills, 95). Recovering these specifications is generally intended to redocument, convert, restructure, maintain or extend old applications. It is also required when developing a Data Administration function that has to know and record the description of all the information resources of the company.

The problem is particularly complex with old and ill-designed applications. In this case, not only can no decent documentation (if any) be relied on, but the lack of systematic methodologies for designing and maintaining them have led to tricky and obscure code. Therefore, reverse engineering has long been recognized as a complex, painful and prone-to-failure activity, so much so that it is simply not undertaken most of the time, leaving huge amounts of invaluable knowledge buried in the programs, and therefore definitively lost.

In information systems, or *data-oriented applications*, i.e. in applications whose central component is a database (or a set of permanent files), the complexity can be broken down by considering that the files or databases can be reverse engineered (almost) independently of the procedural parts.

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

[1] This paper presents some results of the DB-MAIN project. This project is partially supported by the *Région Wallonne*, the *European Union*, and by a consortium comprising ACEC-OSI (Be), ARIANE-II (Be), Banque UCL (Lux), BBL (Be), Centre de recherche public H. Tudor (Lux), CGER (Be), Cockerill-Sambre (Be), CONCIS (Fr), D'Ieteren (Be), DIGITAL, EDF (Fr), EPFL (CH), Groupe S (Be), IBM, OBLOG Software (Port), ORIGIN (Be), Ville de Namur (Be), Winterthur (Be), 3 Suisses (Be). The DB-Process subproject is supported by the *Communauté Française de Belgique*.

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

Therefore, concentrating on reverse engineering the data components of the application first can be much more efficient than trying to cope with the whole application.

The database community considers that there exist two outstanding levels of description of a database or of a consistent collection of files, materialized into two documents, namely its conceptual schema and its logical schema. The first one is an abstract, technology-independent, description of the data, expressed in terms close to the application domain. Conceptual schemas are expressed in some semantics-representation formalisms such as the ERA, NIAM or OMT models. The logical schema describes these data translated into the data model of a specific data manager, such as a commercial DBMS. A logical schema comprises tables, columns, keys, record types, segment types and the like.

The primary aim of database reverse engineering (DBRE) is to recover possible logical and conceptual schemas for an existing database.

1.2. Two Introductory Examples

The real scope of database reverse engineering has sometimes been misunderstood, and presented as merely redrawing the data structures of a database into some DBMS-independent formalism. Many early scientific proposals, and most current CASE tools are limited to the translation process illustrated in Figure 1. In such situations, some elementary translation rules suffice to produce a tentative conceptual schema.

```
create table CUSTOMER (
     CNUM numeric(6)not null,
     CNAME char(24) not null,
     CADDRESS char(48 not null,
     primary key (CNUM))

create table ORDER (
     ONUM char(8) not null,
     CNUM numeric(6) not null,
     ODATE date,
     primary key (ONUM),
     foreign key (CNUM)
     references CUSTOMER)
```

$\Rightarrow$

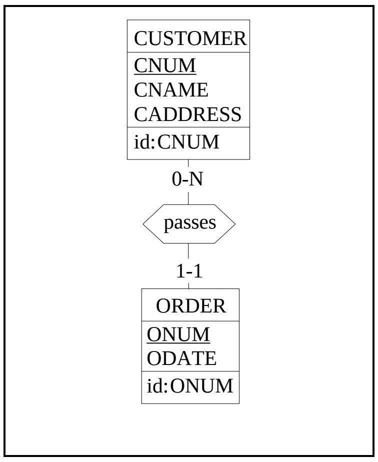


Figure 1. An idealistic view of database reverse engineering

Unfortunately, most situations are actually far more complex. In Figure 2, we describe a very small COBOL fragment from which we intend to extract the semantics underlying the files CF008 and PF0S. By merely analyzing the record structure declarations, as most DBRE CASE tools do at the present time, only schema (a) in Figure 2 can be extracted. It obviously brings little information about the meaning of the data.

```
...
Environment division.
Input-output section.
File control.
select CF008 assign to DSK02:P12
organization is indexed
record key is K1 of REC-CF008-1.

select PF0S assign to DSK02:P27
organization is indexed
record key is K1 of REC-PFOS-1.
...
Data division.
File section.
fd CF008.
01 REC-CF008-1.
   02 K1 pic 9(6).
   02 filler pic X(125).

fd PF0S.
01 REC-PF0S-1.
   02 K1
      03 K11 pic X(9).
      03 filler pic 9(6)
   02 PDATA pic X(180).
   02 PRDATA redefines PDATA.
      03 filler pic 9(4)V99.
...

Working storage section.
01 IN-COMPANY.
   02 CPY-ID pic 9(6).
   02 C-DATA pic X(125).
```

```
01 CPY-DATA.
   02 CNAME pic X(25).
   02 CADDRESS pic X(100).

01 PKEY.
   02 K11 pic X(9).
   02 K12 pic X(6).
...

Procedure division.
...
move zeroes to PKEY.
display "Enter ID :" with no advancing.
accept K11 of PKEY.
move PKEY to K1 of REC-PF0S-1.
read PF0S key K1 on invalid key
  display "Invalid Product ID"
  display PDATA with no advancing.
...
read PF0S.
perform until K11 of K1 > K11 of PKEY
  display "Production:" with no advancing
  display PRDATA with no advancing
  display " tons by " with no advancing
  move K1 of REC-PF0S-1 to PKEY
  move K12 of PKEY to K1 of REC-CF008-1
  read CF008 into IN-COMPANY
    not invalid key
    move C-DATA to CPY-DATA
    display CNAME of CPY-DATA
  end-read.
  read next PF0S
end-perform.
...
```

$\Downarrow$

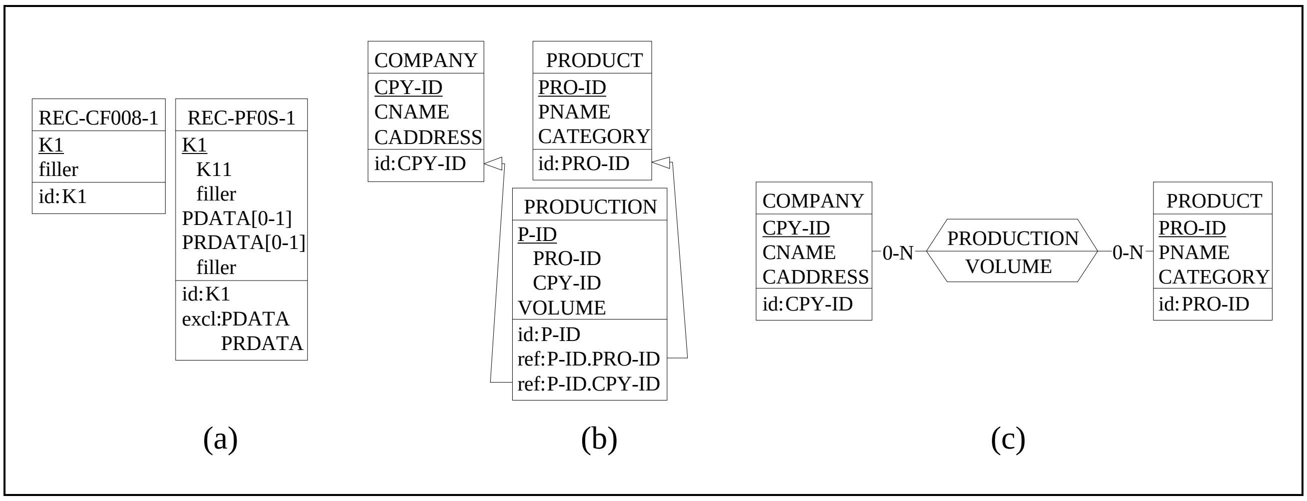


Figure 2. A more realistic view of database reverse engineering. Merely analyzing the data structure declaration statements yields a poor result (a), while further inspection of the procedural code makes it possible to recover a much more explicit schema (b), which can be expressed as a conceptual schema (c).

However, by analyzing the procedural code, the user-program dialogs, and, if needed, the file contents, a more expressive schema can be obtained. For instance, schema (b) can be considered as a refinement of schema (a) resulting from the following reasonings :

- the gross structure of the program suggests that there are two kinds of `REC-PF0S-1` records, arranged into ordered sequences, each comprising one *type-1* record (whose PDATA field is processed before the loop), followed by an arbitrary sequence of *type-2* records (whose PRDATA field is processed in the body of the loop); all the records of such a sequence share the same first 9 characters of the key;
- the processing of *type-1* records shows that the K11 part of key K1 is an identifier, the rest of the key acting as pure padding; the user dialog suggests that *type-1* records describe products; this record type is called PRODUCT, and its key PRO-ID; visual inspection of the contents of the PF0S file could confirm this hypothesis;
- examining the screen contents when running the program shows that PDATA is made of a product name followed by a product category; this interpretation is given by a typical user of the program; this field can then be considered as the concatenation of a PNAME fileld and a CATEGORY field.
- the body of the loop processes the sequence of *type-2* records depending on the current PRODUCT record; they all share the PRO-ID value of their parent PRODUCT record, so that that this 9-digit subfield can be considered as a foreign key to the PRODUCT record;
- the processing of a *type-2* record consists in displaying one line made up of constants and field values; the linguistic structure of this line suggests that it informs us about some *Production* of the current product; the PDATA value is expressed in *tons* (most probably a volume), and seems to be produced *by* some kind of agents described in the file CF008; hence the names PRODUCTION for *type-2* record type and VOLUME for the PRDATA field;
- the *agent* of a production is obtained by using the second part of the key of the PRODUCTION record; therefore, this second part can be considered as a foreign key to the REC-CF008-1 records;
- the name of the field in which the *agent* record is stored suggests that the latter is a company; hence the name COMPANY for this record type, and CPY-ID for its access key;
- the C-DATA field of the COMPANY record type should match the structure of the CPY-DATA variable, which in turn is decomposed into CNAME and CADDRESS.

Refining the initial schema (a) these reasonings result in schema (b), and interpreting these technical data structures into a semantic information model (here some variant of the Entity-relationship model) leads to schema (c).

Despite its small size, this exercise emphasizes some common difficulties of database reverse engineering. In particular, it shows that the declarative statements that define file and record structures can prove a poor source of information. The analyst must often rely on the inspection of other aspects of the application, such as the procedural code, the user-program interaction, the program behaviour, the file contents. This example also illustrates the weaknesses of most data managers which, together with some common programming practices that tend to hide important structures, force the programmer to express essential data properties through procedural code. Finally domain knowledge proves essential to discover and to validate some components of the resulting schema.

1.3. State of the Art

Though reverse engineering data structures is still a complex task, it appears that the current state of the art provides us with concepts and techniques powerful enough to make this enterprise more realistic.

The literature proposes systematic approaches for database schema recovery :

- for standard files : (Casanova, 83), (Nillson, 85), (Davis, 85), (Sabanis, 92)
- for IMS databases : (Navathe, 88), (Winans, 90), (Batini, 92), (Fong, 93)
- for CODASYL databases : (Batini, 92), (Fong, 93), (Edwards, 95)
- for relational databases : (Casanova, 84), (Navathe, 88), (Davis, 88), (Johan, 89), (Markowitz, 90), (Springsteel, 90), (Fonkam, 92), (Batini, 92), (Premerlani, 93), (Chiang, 94), (Shoval, 93), (Petit, 94), (Andersson, 94), (Signore, 94), (Vermeer, 95).

Many of these studies, however, appear to be limited in scope, and are generally based on assumptions about the quality and completeness of the source data structures to be reverse engineered that cannot be relied on in many practical situations. For instance, they often suppose that

- all the conceptual specifications have been translated into data structures and constraints (at least until 1993); in particular, constraints that have been procedurally expressed are ignored;
- the translation is rather straightforward (no tricky representations); for instance, a relational schema is often supposed to be in 4NF[2]; (Premerlani, 93) and (Blaha, 95) are among the only proposals that cope with some non trivial representations, or idiosyncrasies, observed in real world applications; let us mention some of them : nullable primary key attributes, almost unique primary keys, denormalized structures, degradation of inheritance, mismatched referential integrity domains, overloaded attributes, contradictory data;
- the schema has not been deeply restructured for performance objectives or for any other requirements; for instance, record fragmentation or merging for disc space or access time minimization will remain undetected and will be propagated to the conceptual schema;
- a complete and up-to-date DDL schema of the data is available;
- names have been chosen rationally (e.g. a foreign key and the referenced primary key have the same name), so that they can be used as reliable definition of the objects they denote.

In many proposals, it appears that the only databases that can be processed are those that have been obtained by a rigourous database design method. This condition cannot be assumed for most large operational databases, particularly for the oldest ones. Moreover, these proposals are most often dedicated to one data model and do not attempt to elaborate techniques and reasonings common to several models, leaving the question of a general DBRE approach still unanswered.

Since 1992, some authors have recognized that the procedural part of the application programs is an essential source of information on data structures (Joris, 92), (Hainaut, 93a), (Petit, 94), (Andersson, 94), (Signore, 94).

Like any complex process, DBRE cannot be successful without the support of adequate tools called CARE tools[3]. An increasing number of commercial products (claim to) offer DBRE functionalities.

[2] A table is in 4NF *iff* all the non-trivial multivalued dependencies are functional. The BCNF (Boyce-Codd normal form) is weaker but has a more handy definition : a table is in BCNF *iff* each functional determinant is a key.

[3] A CASE tool offering a rich toolset for reverse engineering is often called a CARE (Computer-Aided Reverse Engineering) tool.

Though they ignore many of the most difficult aspects of the problem, those tools provide their users with invaluable help to carry out DBRE more effectively (Rock-Evans, 90).

In (Hainaut, 93a), we proposed the theoretical baselines for a generic, DBMS-independent, DBRE methodology. These baselines have been developed and extended in (Hainaut, 93b) and (Hainaut, 94). The current paper translates these principles into practical requirements DBRE CARE tools should meet, and presents the main aspects and components of a CASE tool dedicated to database applications engineering, and more specifically to database reverse engineering.

1.4. About this paper

The paper is organized as follows. Section 2 is a synthesis of the main problems which occur in practical DBRE, and of a generic DBMS-independent DBRE methodology. Section 3 discusses some important requirements which should be satisfied by future DBRE CARE tools. Section 4 briefly presents a prototype DBRE CASE tool which is intended to address these requirements. The following sections describe in further detail some of the original principles and components of this CASE tool : the specification model and the repository (Section 5), the transformation toolkit Section 6), the user interface (Section 7), the text analyzers and name processor (Section 8), the assistants (Section 9), functional extensibility (Section 10) and methodological control (Section 11). Section 12 evaluates to what extent the tool meets the requirements while Section 13 describes some real world applications of the methodology and of the tool.

The reader is assumed to have some basic knowledge of data management and design. Recent references (Elmasri, 94) and (Date, 94) are suggested for data management, while (Batini, 92) and (Teorey, 94) are recommended for database design.

2. A Generic Methodology for Database Reverse Engineering (DBRE)

The problems that arise when recovering the documentation of the data naturally fall in two categories that are addressed by the two major processes in DBRE, namely *data structure extraction* and *data structure conceptualization* ((Joris, 92), (Sabanis, 92), (Hainaut, 93a)). By *naturally*, we mean that these problems relate to the recovery of two different schemas, and that they require quite different concepts, reasonings and tools. In addition, each of these processes grossly appears as the reverse of a standard database design process (resp. *physical* and *logical* design (Teorey, 94) (Batini, 92)). We will describe briefly these processes and the problems they try to solve. Let us mention however, that partitioning the problems in this way is not proposed by many authors, who prefer proceeding in one step only. In addition, other important processes are ignored in this discussion for simplicity (see (Joris, 92) for instance).

This methodology is generic in two ways. First, its architecture and its processes are largely DMS-independent. Secondly, it specifies what problems have to be solved, and in which way, rather than the order in which the actions must be carried out. Its general architecture, as developed in (Hainaut, 93a), is outlined in Figure 3.

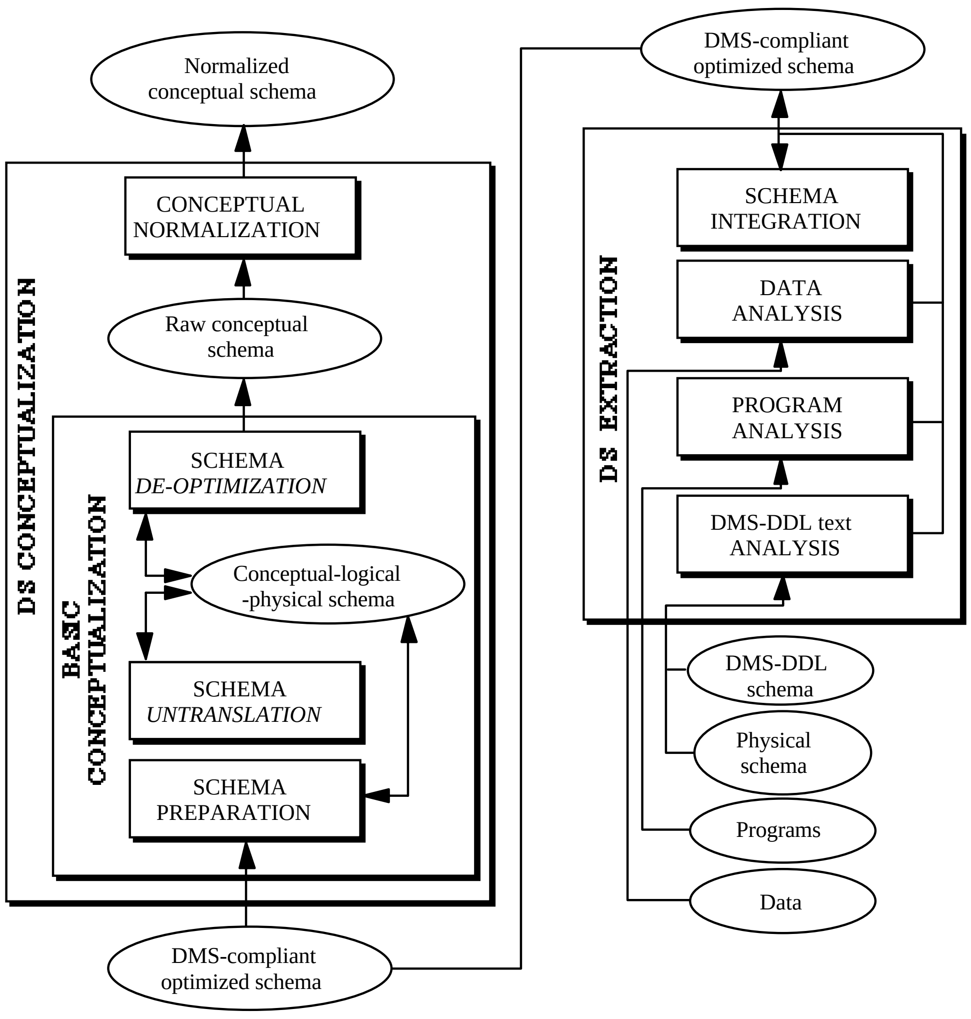


Figure 3. Main components of the generic DBRE methodology. Quite naturally, this reverse methodology is to be read from right to left, and bottom-up !

2.1 Data Structure Extraction

This phase consists in recovering the complete DMS[4] schema, including all the implicit and explicit structures and constraints. True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL texts, etc). Though essential information may be missing from this schema, the latter is a rich starting point that can be refined through further analysis of the other components of the application (views, subschemas, screen and report layouts, procedures, fragments of documentation, database content, program execution, etc).

The problem is much more complex for standard files, for which no computerized description of their structure exist in most cases[5]. The analysis of each source program provides a partial view of the file and record structures only. For most real-world (i.e. non academic) applications, this

4 A Data Management System (DMS) is either a File Management System (FMS) or a Database Management System (DBMS).

5 Though some practices (e.g., disciplined use of COPY or INCLUDE meta-statements to include common data structure descriptions in programs), and some tools (such as data dictionaries) may simulate such centralized schemas.

analysis must go well beyond the mere detection of the record structures declared in the programs. In particular, three problems are encountered, that derive from frequent design practices, namely *structure hiding*, *non declarative structures* and *lost specifications*. Unfortunately, these practices are also common in (true) databases, i.e. those controlled by DBMS, as illustrated by (Premerlani, 93) and (Blaha, 95) for relational databases.

Structure hiding applies to a source data structure or constraint S1, *which could be implemented in the DMS*. It consists in declaring it as another data structure S2 that is more general and less expressive than S1, but that satisfies other requirements such as field reusability, genericity, program conciseness, simplicity or efficiency. Let us mention some examples : a compound/multivalued field in a record type is declared as a single-valued atomic field, a sequence of contiguous fields are merged into a single anonymous field (e.g. as an unnamed COBOL field), a one-to-many relationship type is implemented as a many-to-many link, a referential constraint is not explicitly declared as a foreign key, but is procedurally checked, a relationship type is represented by a foreign key (e.g. in IMS and CODASYL databases).

Non declarative structures are structures or constraints *which cannot be declared in the target DMS*, and therefore are represented and checked by other means, such as procedural sections of the application. Most often, the checking sections are not centralized, but are distributed and duplicated (frequently in different versions), throughout the application programs. Referential integrity in standard files and one-to-one relationship types in CODASYL databases are some examples.

Lost specifications are constructs of the conceptual schema that have not been implemented in the DMS data structures nor in the application programs. This does not mean that the data themselves do not satisfy the lost constraint[6], but the trace of its enforcement cannot be found in the declared data structures nor in the application programs. Let us mention popular examples : uniqueness constraints on sequential files and secondary keys in IMS and CODASYL databases.

Recovering hidden, non declarative and lost specifications is a complex problem, for which no deterministic methods exist so far. A careful analysis of the procedural statements of the programs, of the data flow through local variables and files, of the file contents, of program inputs and outputs, of the user interfaces, of the organizational rules, can accumulate evidence for these specifications. Most often, that evidence must be consolidated by the domain knowledge.

Until very recently, these problems have not triggered much interest in the literature. The first proposals address the recovery of integrity constraints (mainly referential and inclusion) in relational databases through the analysis of SQL queries (Petit, 94), (Andersson, 94), (Signore, 94).

In our generic methodology, the main processes of DATA STRUCTURE EXTRACTION are the following :

- DMS-DDL text ANALYSIS. This rather straighforward process consists in analyzing the data structures declaration statements (in the specific DDL) included in the schema scripts and application programs. It produces a first-cut logical schema.
- PROGRAM ANALYSIS. This process is much more complex. It consists in analyzing the other parts of the application programs, a.o. the procedural sections, in order to detect evidence of additional data structures and integrity constraints. The first-cut schema can therefore be refined following the detection of hidden, non declarative structures.
- DATA ANALYSIS. This refinement process examines the contents of the files and databases in order (1) to detect data structures and properties (e.g. to find the unique fields or the functional

[6] There is no miracle here : for instance, the data are imported, or organizational and behavioural rules make them satisfy these constraints.

dependencies in a file), and (2) to test hypotheses (e.g. "could this field be a foreign key to this file ?"). Hidden, non declarative and lost structures can be found in this way.

- SCHEMA INTEGRATION. When more than one information source has been processed, the analyst is provided with several, generally different, extracted (and possibly refined) schemas. Let us mention some common situations : base tables and views (RDBMS), DBD and PSB (IMS), schema and subschemas (CODASYL), file structures from all the application programs (standard files), etc. The final logical schema must include the specifications of all these partial views, through a *schema integration* process.

The end product of this phase is the complete logical schema. This schema is expressed according to the specific model of the DMS, and still includes possible optimized constructs, hence its name : the *DMS-compliant optimized schema*, or *DMS schema* for short.

The current DBRE CARE tools offer only limited *DMS-DDL text ANALYSIS* functionalities. The analyst is left without help as far as *PROGRAM ANALYSIS*, *DATA ANALYSIS* and *SCHEMA INTEGRATION* processes are concerned. The DB-MAIN tool is intended to address all these processes and to improve the support that analysts are entitled to expect from CARE tools.

2.2 Data Structure Conceptualization

This second phase addresses the conceptual interpretation of the DMS schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. It consists of two sub-processes, namely *Basic conceptualization* and *Conceptual normalization*. The reader will find in (Hainaut, 93b) a more detailed description of these processes, which rely heavily on schema restructuring techniques (or schema transformations).

- BASIC CONCEPTUALIZATION. The main objective of this process is to extract all the relevant semantic concepts underlying the logical schema. Two different problems, requiring different reasonings and methods, have to be solved : *schema untranslation* and *schema de-optimization*. However, before tackling these problems, we often have to *prepare* the schema by cleaning it.

  SCHEMA PREPARATION. The schema still includes some constructs, such as files and access keys, which may have been useful in the Data Structure Extraction phase, but which can now be discarded. In addition, translating names to make them more meaningful (e.g. substitute the file name for the record name), and restructuring some parts of the schema can prove useful before trying to interpret them.

  SCHEMA UNTRANSLATION. The logical schema is the technical translation of conceptual constructs. Through this process, the analyst identifies the traces of such translations, and replaces them by their original conceptual construct. Though each data model can be assigned its own set of translating (and therefore of untranslating) rules, two facts are worth mentioning. First, the data models can share an important subset of translating rules (e.g. COBOL files and SQL structures). Secondly, translation rules considered as specific to a data model are often used in other data models (e.g. foreign keys in IMS and CODASYL databases). Hence the importance of generic approaches and tools.

  SCHEMA DE-OPTIMIZATION. The logical schema is searched for traces of constructs designed for optimization purposes. Three main families of optimization techniques should be considered : denormalization, structural redundancy and restructuring (Hainaut, 93b).

- CONCEPTUAL NORMALIZATION. This process restructures the basic conceptual schema in order to give it the desired qualities one expects from any final conceptual schema, e.g. expressiveness, simplicity, minimality, readability, genericity, extensibility. For instance, some entity types are replaced by relationship types or by attributes, is-a relations are made explicit, names are standardized, etc. This process is borrowed from standard DB design methodologies (Batini, 92), (Teorey, 94), (Rauh, 95).

All the proposals published so far address this phase, most often for specific DMS, and for rather simple schemas (e.g. with no implementation tricks). They generally propose elementary rules and heuristics for the *SCHEMA UNTRANSLATION* process and to some extent for *CONCEPTUAL NORMALIZATION*, but not for the more complex *DE-OPTIMIZATION* phase. The DB-MAIN CARE tool has been designed to address all these processes in a flexible way.

2.3 Summary of the Limits of the State of the Art in CARE Tools

The methodological framework developed in Sections 2.1 and 2.2 can be specialized according to a specific DMS and according to specific development standards. For instance (Hainaut, 93b) suggests specialized versions of the *CONCEPTUALIZATION* phase for SQL, COBOL, IMS, CODASYL and TOTAL/IMAGE databases.

It is interesting to use this framework as a reference process model against which existing methodologies can be compared, in particular, those which underlie the current CARE tools (Figure 4). The conclusions can be summarized as follows :

- DATA STRUCTURE EXTRACTION : current CARE tools are limited to parsing DMS-DDL schemas only (DMS-DDL text ANALYSIS). All the other sources are ignored, and must be processed manually. For instance, these tools are unable to collect the multiple views of a COBOL application, and to integrate them to produce the global COBOL schema. A user of a popular CARE tool tells us "*how he spent several weeks, cutting and pasting hundreds of sections of programs, to build an artificial COBOL program in which all the files and records were fully described. Only then was the tool able to extract the file data structures*".
- DATA STRUCTURE CONCEPTUALIZATION : current CARE tools focus mainly on untranslation (SCHEMA UNTRANSLATION) and offer some restructuring facilities (CONCEPTUAL NORMALIZATION), though these processes often are merged. Once again, some strong naming conventions must often be satisfied for the tools to help. For instance, a foreign key and the referenced primary key must have the same names. All performance-oriented constructs, as well as most non standard database structures, (see (Premerlani, 93) and (Blaha, 95) for instance) are completely beyond the scope of these tools.

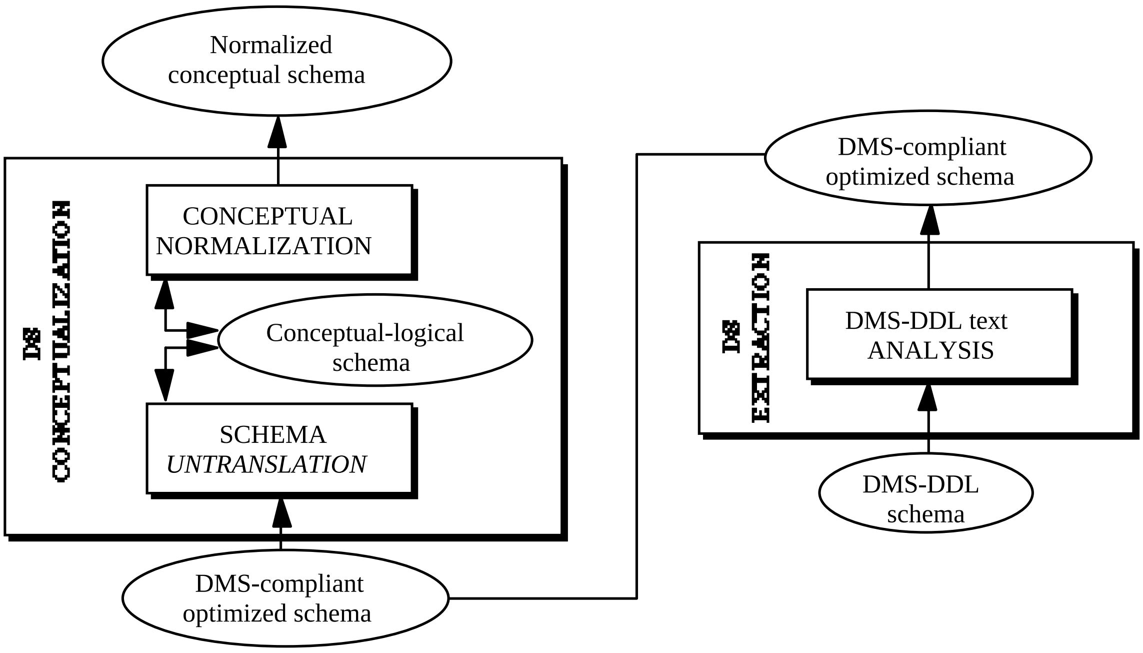


Figure 4. Simplified DBRE methodology proposed by most current CARE tools.

3. Requirements for a DBRE CARE Tool

This section states some of the most important requirements an ideal DBRE support environment (or CARE tool) should meet. These requirements are induced by the analysis of the specific characteristics of the DBRE processes. They also derive from reverse engineering files and databases, often by hand or with very basic text editing tools, of a dozen actual applications.

Flexibility

Observation : The very nature of the RE activities differs from that of more standard engineering activities. Reverse engineering a software component, and particularly a database, is basically an *exploratory* and often *unstructured* activity. Some important aspects of higher level specifications are *discovered* (sometimes by chance), and not deterministically inferred from the operational ones.

Requirements : The tool must allow the user to follow flexible working patterns, including unstructured ones. It should be methodology-neutral[7] unlike forward engineering tools. In addition, it must be highly interactive.

Extensibility

Observation : RE appears as a learning process; each RE project often is a new problem of its own, requiring specific reasonings and techniques.

Requirements : Specific functions should be easy to develop, even for one-shot use.

[7] But methodology-aware if *design recovery* is intended. This aspect has been developed in [Hai,94], and will be evoked in section 11.

Source multiplicity

Observation : RE requires a great variety of information sources : data structure, data (from files, databases, spreadsheets, etc), program text, program execution, program output, screen layout, CASE repository and Data dictionary contents, paper or computer-based documentation, interview, workflow and dataflow analysis, domain knowledge, etc.

Requirements : The tool must include browsing and querying interfaces with these sources. Customizable functions for automatic and assisted specification extraction should be available for each of them.

Text analysis

Observation : More particularly, database RE requires browsing through huge amounts of text, searching them for specific patterns (e.g. programming *clichés* (Selfridge, 93)), following static execution paths and dataflows, extracting program slices (Weizer, 84).

Requirements : The CARE tool must provide sophisticated text analysis processors. The latter should be language independent, easy to customize and to program, and tightly coupled with the specification processing functions.

Name processing

Observation : Object names in the operational code are an important knowledge source. Frustratingly enough, these names often happen to be meaningless (e.g. REC-001-R08, I-087), or at least less informative than expected (e.g. INV-QTY, QOH, C-DATA), due to the use of strict naming conventions. Many applications are multilingual[8], so that data names may be expressed in several languages. In addition, multi-programmer development often induces non consistent naming conventions.

Requirements : The tool must include sophisticated name analysis and processing functions.

Links with other CASE processes

Observation : RE is seldom an independent activity. For instance, (1) forward engineering projects frequently include reverse engineering of some existing components, (2) reverse engineering share important processes with forward engineering (e.g. conceptual normalization), (3) reverse engineering is a major activity in broader processes such as migration, reengineering and data administration.

Requirements : A CARE tool must offer a large set of functions, including those which pertain to forward engineering.

Openness

Observation : There is (and probably will be) no available tool that can satisfy all corporate needs in application engineering. In addition, companies usually already make use of one or, most often, several CASE tools, software development environments, DBMS, 4GL or DDS.

Requirements : A CARE tool must communicate easily with the other development tools (e.g. via integration hooks, communications with a common repository or by exchanging specifications through a common format).

[8] For instance, Belgium commonly uses three legal languages, namely Dutch, French and German. As a consequence, English is often used as a *de facto* common language.

Flexible specification model

Observation : As in any CAD activity, RE applies on incomplete and inconsistent specifications. However, one of its characteristics makes it intrinsically different from design processes : at any time, the current specifications may include components from different abstraction levels. For instance, a schema in process can include record types (physical objects) as well as entity types (conceptual objects).

Requirements : The specification model must be *wide-spectrum*, and provides artifacts for components of different abstraction levels.

Genericity

Observation : Tricks and implementation techniques specific to some data models have been found to be used in other data models as well (e.g. foreign keys are frequent in IMS and CODASYL databases). Therefore, many RE reasonings and techniques are common to the different data models used by current applications.

Requirements : The specification model and the basic techniques offered by the tool must be DMS-independent, and therefore highly generic.

Multiplicity of views

Observation : The specifications, whatever their abstraction level (e.g. physical, logical or conceptual), are most often huge and complex, and need to be examined and browsed through in several ways, according to the nature of the information one tries to obtain.

Requirements : The CARE tool must provide several ways of viewing both source texts and abstract structures (e.g. schemas). Multiple textual and graphical views, summary and fine-grained presentations must be available.

Rich transformation toolset

Observation : Actual database schemas may include constructs intended to represent conceptual structures and constraints in non standard ways, and to satisfy non functional requirements (performance, distribution, modularity, access control, etc). These constructs are obtained through schema restructuration techniques.

Requirements : The CARE tool must provide a rich set of schema transformation techniques. In particular, this set must include operators which can *undo* the transformations commonly used in practical database designs.

Traceability

Observation : A DBRE project includes at least three sets of documents : the operational descriptions (e.g. DDL texts, source program texts), the logical schema (DMS-compliant) and the conceptual schema (DMS-independent). The forward and backward mappings between these specifications must be precisely recorded. The forward mapping specifies how each conceptual (or logical) construct has been implemented into the operational (or logical) specifications, while the backward mapping indicates of which conceptual (or logical) construct each operational (or logical) construct is an implementation.

Requirements : The repository of the CARE tool must record all the links between the schemas at the different levels of abstraction. More generally, the tool must ensure the *traceability* of the RE processes.

4. The DB-MAIN CASE Tool

The DB-MAIN database engineering environment is a result of a R & D project initiated in 1993 by the DB research unit of the Institute of Informatics, University of Namur. This tool is dedicated to *database applications engineering*, and its scope encompasses, but is much broader than, reverse engineering alone. In particular, its ultimate objective is to assist developers in database design (including full control of logical and physical processes), database reverse engineering, database application reengineering, maintenance, migration and evolution. Further detail on the whole approach can be found in (Hainaut, 94).

As far as DBRE support is concerned, the DB-MAIN CASE tool has been designed to address as much as possible the requirements developed in the previous section.

As a wide-scope CASE tool, DB-MAIN includes the usual functions needed in database analysis and design, e.g. entry, browsing, management, validation and transformation of specifications, as well as code and report generation. However, the rest of this paper, namely Sections 5 to 11, will concentrate only on the main aspects and components of the tool which are directly related to DBRE activities. In Section 5 we describe the way schemas and other specifications are represented in the repository. The tool is based on a general purpose transformational approach which is described in Section 6. Viewing the specifications from different angles and in different formats is discussed in Section 7. In Section 8, various tools dedicated to text and name processing and analysis are described. Section 9 presents some expert modules, called *assistant,* which help the analyst in complex processing and analysis tasks. DB-MAIN is an extensible tool which allows its users to build new functions through the *Voyager-2* tool development language (Section 10). Finally, Section 11 evokes some aspects of the tool dedicated to methodological customization, control and guidance.

In Section 12, we will reexamine the requirements described in Section 3, and evaluate to what extent the DB-MAIN tool meets them.

5. The DB-MAIN Specification Model and Repository

The repository collects and maintains all the information related to a project. We will limit the presentation to the data aspects only. Though they have strong links with the data structures in DBRE, the specification of the other aspects of the applications, e.g. processing, will be ignored in this paper. The repository comprises three classes of information :

- a structured collection of schemas and texts used and produced in the project,
- the specification of the methodology followed to conduct the project,
- the history (or trace) of the project.

We will ignore the two latter classes, which are related to methodological control and which will be evoked briefly in Section 11.

A *schema* is a description of the data structures to be processed, while a *text* is any textual material generated or analyzed during the project (e.g. a program or an SQL script). A project usually comprises many (i.e. dozens to hundreds of) schemas. The schemas of a project are linked through specific relationships; they pertain to the methodological control aspects of the DB-MAIN approach, and will be ignored in this section.

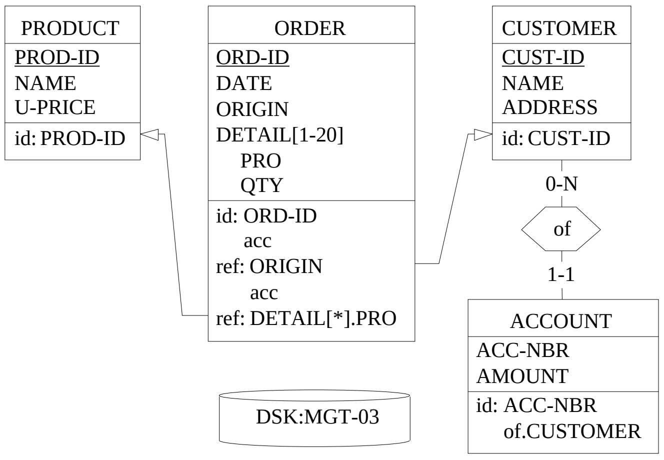


Figure 5. A typical data structure schema during reverse engineering. This schema includes conceptualized objects (PRODUCT, CUSTOMER, ACCOUNT, of), logical objects (record type ORDER, with single-valued and multivalued foreign keys) and physical objects (access keys ORDER.ORD-ID and ORDER.ORIGIN; file DSK:MGT-03)

A schema is made up of specification constructs which can be classified into the usual three abstraction levels. The DB-MAIN specification model includes the following concepts (Hainaut, 92a).

The *conceptual constructs* are intended to describe abstract, machine-independent, semantic structures. They include the notions of entity types (with/without attributes; with/without identifiers), of super/subtype hierarchies (single/multiple inheritance, total and disjoint properties), and of relationship types (binary/N-ary; cyclic/acyclic), whose roles are characterized by min-max cardinalities and optional names; a role can be defined on one or several entity-types. Attributes can be associated with entity and relationship types; they can be single-valued or multivalued, atomic or compound. Identifiers (or keys), made up of attributes and/or roles, can be associated with an entity type, a relationship type and a multivalued attribute. Various constraints can be defined on these objects : inclusion, exclusion, coexistence, at-least-one, etc.

The *logical constructs* are used to describe schemas compliant with DMS models, such as relational, CODASYL or IMS schemas. They comprise, among others, the concepts of record types (or table, segment, etc), fields (or columns), referential constraints, and redundancy.

The *physical constructs* describe implementation aspects of the data which are related to such criteria as the performance of the database. They make it possible to specify files, access keys (index, calc key, etc), physical data types, bag and list multivalued attributes, and other implementation details.

In database engineering, as discussed in Section 2, a schema describes a fragment of the data structures at a given level of abstraction. In reverse engineering, an *in progress* schema may even include constructs at different levels of abstraction. Figure 5 presents a schema which includes conceptual, logical and physical constructs. Ultimately, this schema will be completely conceptualized through the interpretation of the logical and physical objects.

Besides these concepts, the repository includes some generic objects which can be customized according to specific needs. In addition, annotations can be associated with each object. These annotations can include semi-formal properties, made of the property name and its value, which can be interpreted by *Voyager-2* functions (see Section 10). These features provide dynamic extensibility of the repository. For instance, new concepts such as *organizational units*, *servers*, or *geographic sites* can be represented by specializing the generic objects, while *statistics* about entity populations, the *gender* and *plural* of the object names can be represented by semi-formal attributes. The contents of the repository can be expressed as a pure text file through the ISL language, which provides import-export facilities between DB-MAIN and its environment.

6. The Transformation Toolkit

The desirability of the transformational approach to software engineering is now widely recognized. According to (Fickas, 85) for instance, *the process of developing a program [can be] formalized as a set of transformations*. This approach has been put forward in database engineering by an increasing number of authors since several years, either in research papers, or in text books and, more recently, in several CASE tools (Hainaut, 92a) (Rosenthal, 94). Quite naturally, schema transformations have found their way into DBRE as well (Hainaut, 93a) (Hainaut, 93b). The transformational approach is the cornerstone of the DB-MAIN approach (Hainaut, 81) (Hainaut, 91a) (Hainaut, 93b) (Hainaut, 94) and CASE tool (Hainaut, 92a) (Joris, 92) (Hainaut, 94). A formal presentation of this concept can be found in (Hainaut, 91a), (Hainaut, 95) and (Hainaut, 96).

Roughly speaking, a schema transformation consists in deriving a target schema S' from a source schema S by some kind of local or global modification. Adding an attribute to an entity type, deleting a relationship type, and replacing a relationship type by an equivalent entity type, are three examples of schema transformations. Producing a database schema from another schema can be carried out through selected transformations. For instance, normalizing a schema, optimizing a schema, producing an SQL database or COBOL files, or reverse engineering standard files and CODASYL databases can be described mostly as sequences of schema transformations. Some authors propose schema transformations for selected design activities (Navathe, 80) (Kobayashi, 86) (Kozaczynsky, 87) (Rosenthal, 88) (Batini, 92) (Rauh, 95) (Halpin, 95). Moreover, some authors claim that the whole database design process, together with other related activities, can be described as a chain of schema transformations (Batini, 93) (Hainaut, 93b) (Rosenthal, 94). Schema transformations are essential to define formally forward and backward mappings between schemas, and particularly between conceptual structures and DMS constructs (i.e. traceability).

A special class of transformations is of particular importance, namely the *semantics-preserving* transformations, also called *reversible* since each of them is associated with another semantics-preserving transformation called its *inverse*. Such a transformation ensures that the source and target schemas have the same semantic descriptive power. In other words any situation of the application domain that can be modelled by an instance of one schema can be described by an instance of the other. If we can produce a relational schema from a conceptual schema by applying reversible transformations only, then both schemas will be equivalent by construction, and no semantics will be lost in the translation process. Conversely, if the interpretation of a relational schema, i.e. its conceptualization (Section 2.2), can be performed by using reversible transformations, the resulting conceptual schema will be semantically equivalent to the source schema. An in-depth discussion of the concept of specification preservation can be found in (Hainaut, 95) and (Hainaut, 96).

To illustrate this concept, we will outline informally three of the most popular transformation techniques, called **mutations** (type changing) used in database design. As a consequence, their inverse will be used in reverse engineering. To simplify the presentation, each transformation and its inverse are described in one figure, in which the direct transformation is read from left to right, and its inverse from right to left.

Figure 6 shows graphically how a relationship type can be replaced by an equivalent entity type, and conversely. The technique can be extended to N-ary relationship types.

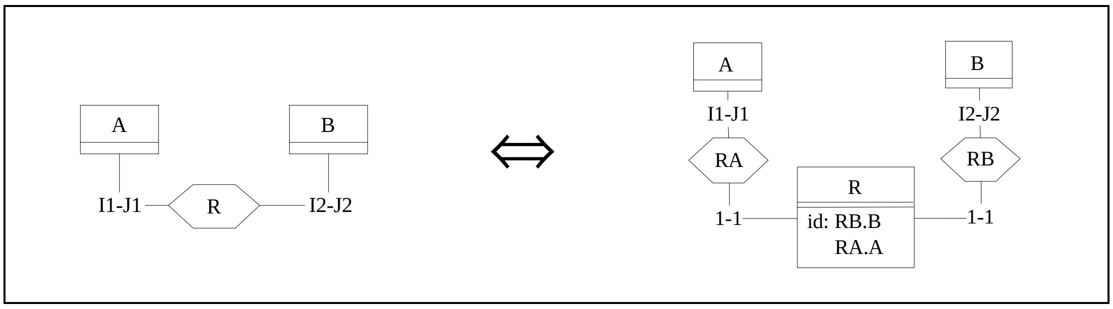


Figure 6. Transforming a relationship type into an entity type, and conversely.

Another widely used transformation replaces a binary relationship type by a *foreign key* (Figure 7), which can be either multivalued (J > 1) or single-valued (J = 1).

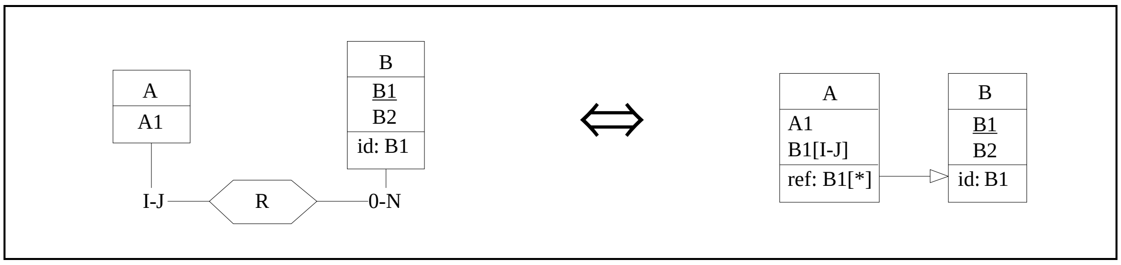


Figure 7. Relationship-type R is represented by foreign key B1, and conversely.

The third standard technique transforms an attribute into an entity type. It comes in two flavours, namely *instance representation* (Figure 8a), in which each instance of attribute A2 in each A entity is represented by an EA2 entity, and *value representation* (Figure 8b), in which each distinct value of A2, whatever the number of its instances, is represented by one EA2 entity.

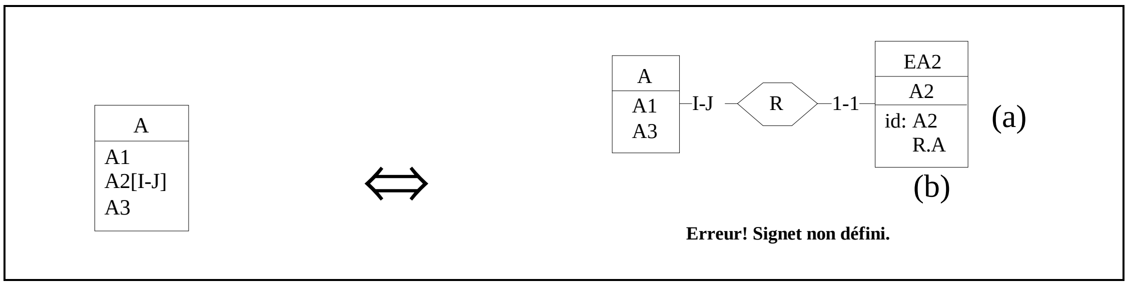


Figure 8. Transformation of an attribute into an entity type : (a) by explicit representation of its **instances**, (b) by explicit representation of its distinct **values**.

DB-MAIN proposes a three-level transformation toolset that can be used freely, according to the skill of the user and the complexity of the problem to be solved. These tools are neutral and generic, in that they can be used in any database engineering process. As far as DBRE is concerned, they are mainly used in the Data Structure Conceptualization (Section 2.2).

- **Elementary transformations**. The selected transformation is applied to the selected object :

  `apply transformation T to current object O`

  With these tools, the user keeps full control of the schema transformation. Indeed, similar situations can often be solved by different transformations; e.g. a multivalued attribute can be transformed in a dozen ways. Figure 9 illustrates the dialog box for the Split/Merge of an entity type. The current version of DB-MAIN proposes a toolset of about 25 elementary transformations.

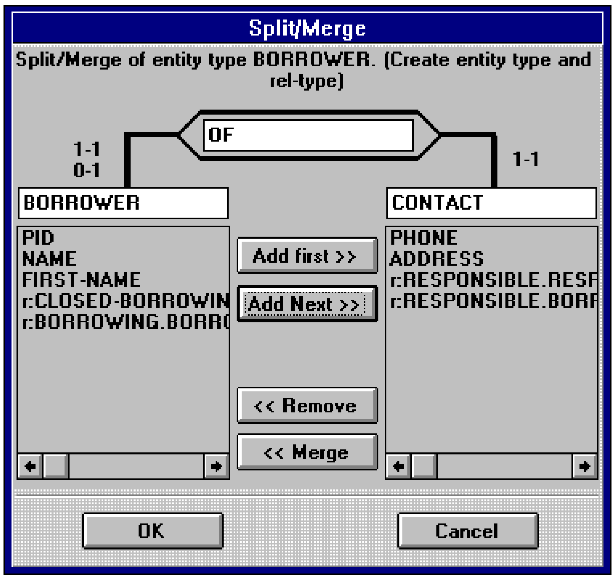


Figure 9. The dialog box of the Split/Merge transformation through which the analyst can either extract some components from the master entity type (left), or merge two entity types, or migrate components from an entity type to another.

- **Global transformations**. A selected elementary transformation is applied to all the objects of a schema which satisfy a specified precondition :

  `apply transformation T to the objects that satisfy condition P`

  DB-MAIN offers some predefined global transformations, such as : *replace all one-to-many relationship types by foreign keys* or *replace all multivalued attributes by entity types*. However, the analyst can define its own toolset through the Transformation Assistant described in Section 9.

- **Model-driven transformations**. All the constructs of a schema that violate a given model M are transformed in such a way that the resulting schema complies with M:

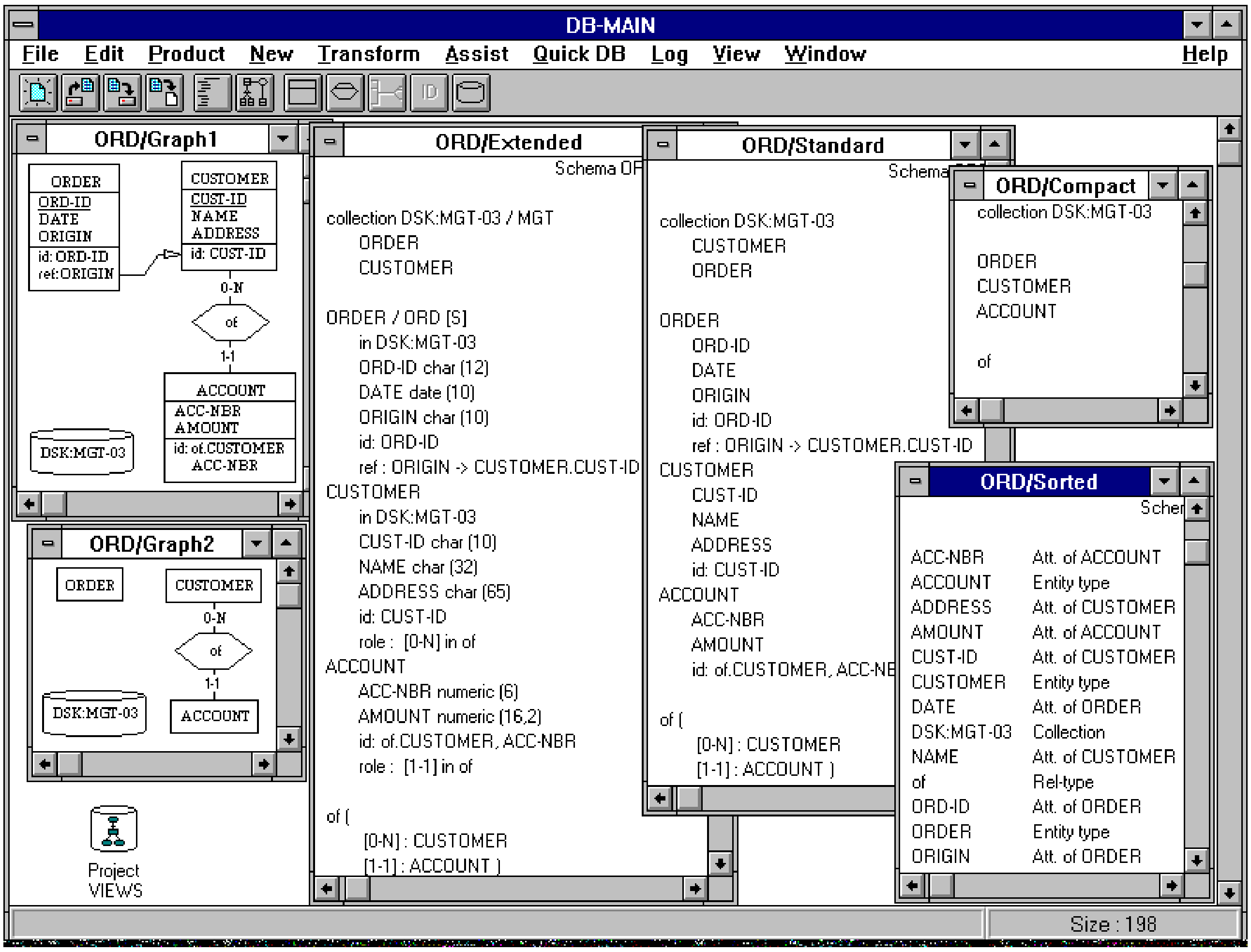
  `apply the transformation plan which makes the current schema satisfy model M`

  Such an operator is defined by a *transformation plan*, which is a sort of algorithm comprising global transformations, which is proved (or assumed) to make any schema comply with M. A model-driven transformation implements formal techniques or heuristics applicable in such major engineering processes as normalization, model translation or untranslation, and conceptualization. Here too, DB-MAIN offers a dozen predefined model-based transformations such as relational, CODASYL, and COBOL translation, untranslation from these models and conceptual normalization. The analyst can define its own transformation plans, either through the scripting facilities of the Transformation Assistant, or, for more complex problems, through the development of *Voyager-2* functions (Section 10).

A more detailed discussion of these three transformation modes can be found in (Hainaut, 92a) and (Hainaut, 95).

7. The User Interface

The user interaction uses a fairly standard GUI. However, DB-MAIN offers some original options which deserve being discussed.



```
ORD/Extended
                                   Schema OR
collection DSK:MGT-03 / MGT
     ORDER
     CUSTOMER

ORDER / ORD [S]
     in DSK:MGT-03
     ORD-ID char (12)
     DATE date (10)
     ORIGIN char (10)
     id: ORD-ID
     ref : ORIGIN -> CUSTOMER.CUST-ID
CUSTOMER
     in DSK:MGT-03
     CUST-ID char (10)
     NAME char (32)
     ADDRESS char (65)
     id: CUST-ID
     role :  [0-N] in of
ACCOUNT
     ACC-NBR numeric (6)
     AMOUNT numeric (16,2)
     id: of.CUSTOMER, ACC-NBR
     role :  [1-1] in of

of (
      [0-N] : CUSTOMER
      [1-1] : ACCOUNT )
```

```
ORD/Standard
                                   Schema
collection DSK:MGT-03
     CUSTOMER
     ORDER

ORDER
     ORD-ID
     DATE
     ORIGIN
     id: ORD-ID
     ref : ORIGIN -> CUSTOMER.CUST-ID
CUSTOMER
     CUST-ID
     NAME
     ADDRESS
     id: CUST-ID
ACCOUNT
     ACC-NBR
     AMOUNT
     id: of.CUSTOMER, ACC-NE

of (
      [0-N] : CUSTOMER
      [1-1] : ACCOUNT )
```

```
ORD/Compact
collection DSK:MGT-03

ORDER
CUSTOMER
ACCOUNT

of
```

```
ORD/Sorted
                          Scher
ACC-NBR      Att. of ACCOUNT
ACCOUNT      Entity type
ADDRESS      Att. of CUSTOMER
AMOUNT       Att. of ACCOUNT
CUST-ID      Att. of CUSTOMER
CUSTOMER     Entity type
DATE         Att. of ORDER
DSK:MGT-03   Collection
NAME         Att. of CUSTOMER
of           Rel-type
ORD-ID       Att. of ORDER
ORDER        Entity type
ORIGIN       Att. of ORDER
```

Figure 10. Six different views of the same schema

Browsing through, processing, and analyzing, large schemas requires an adequate presentation of the specifications. It quickly appears that more than one way of viewing them is necessary. For instance, a graphical representation of a schema allows an easy detection of certain structural patterns, but is useless when analyzing name structures to detect similarities as is common in the DATA STRUCTURE EXTRACTION process (Section 2.1). DB-MAIN currently offers six ways of presenting a schema. Four of these views use a hypertext technique : *compact* (sorted list of entity type, relationship type and file names), *standard* (same + attributes, roles and constraints), *extended* (same + domains, annotations, ET-RT cross-reference) and *sorted* (sorted list of all the object names). Two views are graphical : *full* and *compact* (no attributes and no constraints). All of them are illustrated in figure 10.

Switching from one view to another is immediate, and any object selected in a view is still current when another view is chosen. Any relevant operator can be applied to an object, whatever the view through which it is presented. In addition, the text-based views makes it possible to navigate from entity types to relationship types and conversely through hypertext links.

8. Text Analysis and Processing

Analyzing and processing various kinds of texts are basic activities in two specific processes, namely `DMS-DDL text ANALYSIS` and `PROGRAM ANALYSIS`.

The first process is rather simple, and can be carried out by automatic extractors which analyze the data structure declaration statements of programs and build the corresponding abstract objects in the repository. DB-MAIN currently offers built-in standard parsers for COBOL, SQL, CODASYL, IMS, and RPG, but other parsers can be developed in *Voyager-2* (Section 10).

To address the requirements of the second process, through which the preliminary specifications are refined from evidence found in programs or in other textual sources, DB-MAIN includes a collection of program analysis tools comprising, at the present time, an interactive pattern-matching engine, a dataflow diagram inspector and a program slicer. The main objective of these tools is to contribute to program understanding as far as data manipulation is concerned.

The *pattern-matching* function allows searching text files for definite patterns or *clichés* expressed in PDL, a *Pattern Definition Language*. As an illustration, we will describe one of the most popular heuristics to detect an implicit foreign key in a relational schema. It consists in searching the application programs for some forms of SQL queries which evoke the presence of an undeclared foreign key (Signore, 94) (Andersson, 94) (Petit, 94). The principle is simple : most multi-table queries use primary/foreign key joins. For instance, considering that column CNUM has been recognized as a candidate key of table CUSTOMER, the following query suggests that column CUST in table ORDER may be a foreign key to CUSTOMER :

```
select CUSTOMER.CNUM, CNAME, DATE
from   ORDER, CUSTOMER
where  ORDER.CUST = CUSTOMER.CNUM
```

More generally, any SQL expression that looks like :

```
select ...
from   ... T1,...T2 ...
where  ... T1.C1 = T2.C2 ...
```

may suggest that C1 is a foreign key to table T2 or C2 a foreign key to T1. Of course, this evidence would be even stronger if we could prove that C2 - resp. C1 - is a key of its table. This is just what Figure 11 translates more formally in PDL.

The SQL generic patterns

```
T1 ::= table-name
T2 ::= table-name
C1 ::= column-name
C2 ::= column-name
join-qualif ::=
     begin-SQL
       select select-list
       from  ! {@T1 ! @T2 | @T2 ! @T1}
       where ! @T1"."@C1 _ "=" _ @T2"."@C2 !
     end-SQL
```

The COBOL/DB2 specific patterns

```
AN-name ::= [a-zA-Z][-a-zA-Z0-9]
table-name ::= AN-name
column-name ::= AN-name
_ ::= ({"/n"|"/t"|" "})+
- ::= ({"/n"|"/t"|" "})*
begin-SQL ::= {"exec"|"EXEC"}
             _{"sql"|"SQL"}_
end-SQL ::=  _{"end"|"END"}
              {"-exec"|"-EXEC"}-"."
select ::= {"select"|"SELECT"}
from ::= {"from"|"FROM"}
where ::= {"where"|"WHERE"}
select-list ::= any-but(from)
! ::= any-but({where|end-SQL})
      {","|"/n"|"/t"|" "}
```

Figure 11. Generic and specific patterns for foreign key detection in SQL queries. In the specific patterns, "_" designates any non-empty separator, "-" any separator, and "AN-name" any alphanumeric string beginning with a letter. The "any-but(E)" function identifies any string not including expression E. Symbols "+", "*", "/t", "/n", "|" and "a-z" have their usual *grep* or *BNF* meaning.

This example illustrates two essential features of PDL and of its engine.

1. A set of patterns can be split into two parts (stored in different files). When a *generic pattern file* is opened, the unresolved patterns are to be found in the specified *specific pattern file*. In this example, the generic patterns define the skeleton of an SQL query, which is valid for any RDBMS and any host language, while the specific patterns complement this skeleton by defining the *COBOL/DB2* API conventions. Replacing the latter will allow processing, e.g., *C/ORACLE* programs.
2. A pattern can include variables, the name of which is prefixed with @. When such a pattern is instantiated in a text, the variables are given a value which can be used, e.g., to automatically update the repository.

The pattern engine can analyze external source files, as well as textual descriptions stored in the repository (where, for instance, the extractors store the statements they do not understand, such as comments, SQL `trigger` and `check`). These texts can be searched for visual inspection only, but pattern instantiation can also trigger DB-MAIN actions. For instance, if a procedure such as that presented in Figure 11 (creation of a referential constraint between column C2 and table T1) is associated with this pattern, this procedure can be executed automatically (under the analyst's control) for each instantiation of the pattern. In this way, the analyst can build a powerful custom tool which detects foreign keys in queries and which adds them to the schema automatically.

The *dataflow inspector* builds a graph whose nodes are the variables of the program to be analyzed, and the edges are relationships between these variables. These relationships are defined by selected PDL patterns. For instance, the following COBOL rules can be used to build a graph in which two nodes are linked if their corresponding variables appear simultaneously in a simple assignment statement, in a redefinition declaration, in an indirect write statement or in comparisons :

```
var_1 ::= cob_var;
var_2 ::= cob_var;
```

```
move ::= "MOVE" - @var_1 - "TO" - @var_2 ;
redefines ::= @var_1 - "REDEFINES" - @var_2 ;
write ::= "WRITE" - @var_1 - "FROM" @var_2 ;
if ::= "IF" - @var_1 - rel_op - @var_2 ;
if_not ::= "IF" - @var_1 - "NOT" - rel_op - @var_2 ;
```

This tool can be used to solve structure hiding problems such as the decomposition of anonymous fields and procedurally controlled foreign keys, as illustrated in Figure 2.

The first experiments have quickly taught us that pattern-matching and dataflow inspection work fine for small programs and for locally concentrated patterns, but can prove difficult to use for large programs. For instance, a pattern made of a dozen statements can span several thousands lines of code. With this problem in mind, we have developed a variant of *program slicer* (Weiser, 84), which, given a program P, generates a new program P' defined as follows. Let us consider a point S in P (generally a statement) and an object O (generally a variable) of P. The program slice of P for O at S is the smallest subset P' of P whose execution will give O the same state at S as would the execution of P in the same environment. Generally P' is a very small fragment of P, and can be inspected much more efficiently and reliably, both visually and with the help of the analysis tools described above, than its source program P. One application in which this program slicer has proved particularly valuable is the analysis of the statements contributing to the state of a record when it is written in its file.

DB-MAIN also includes a *name processor* which can transform selected names in a schema, or in selected objects of a schema, according to substitution patterns. Here are some examples of such patterns :

`"^C-" -> "CUST-"`	replaces all prefixes "`C-`" with the prefix "`CUST-`";
`"DATE" -> "TIME"`	replaces each substring "`DATE`", whatever its position, with the substring "`TIME`";
`"^CODE$" -> "REFERENCE"`	replaces all the names "`CODE`" with the new name "`REFERENCE`".

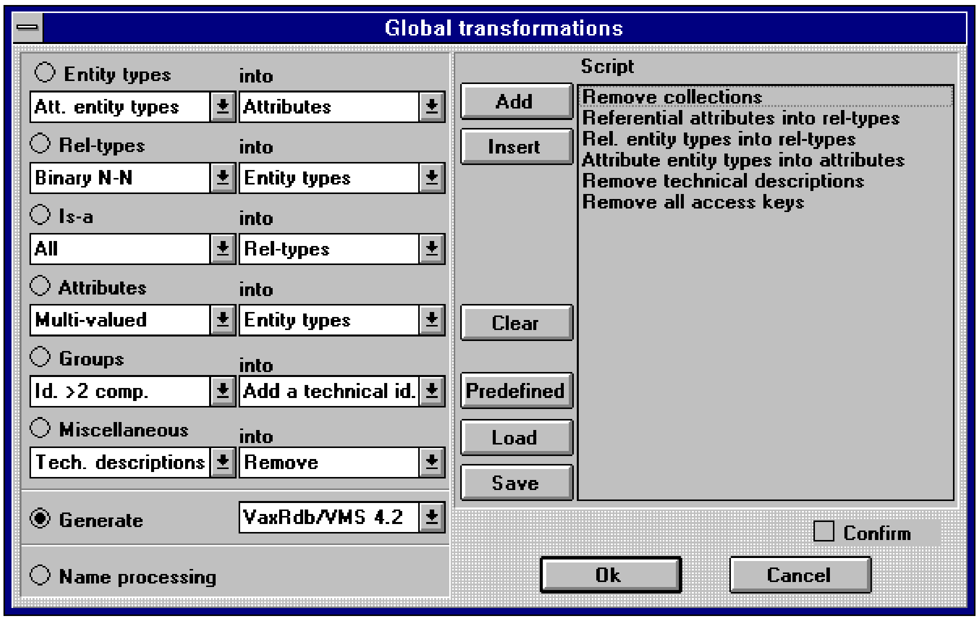
In addition, it proposes case transformation : lower-to-upper, upper-to-lower, capitalize and remove accents. These parameters can be saved as a *name processing script*, and reused later.

9. The Assistants

An assistant is a higher-level solver dedicated to coping with a special kind of problems, or performing specific activities efficiently. It gives access to the basic toolboxes of DB-MAIN, but in a controlled and intelligent way.

The current version of DB-MAIN includes three general purpose assistants which can support, among others, the DBRE activities, namely the *Transformation* assistant, the *Schema Analysis* assistant and the *Text Analysis* assistant. These processors offer a collection of built-in functions that can be enriched by user-defined functions developed in *Voyager-2* (Section 10).

The *Transformation Assistant* (Figure 12) allows applying one or several transformations to selected objects. Each operation appears as a problem/solution couple, in which the problem is defined by a pre-condition (e.g. the objects are *the many-to-many relationship types* of the current schema), and the solution is an action resulting in eliminating the problem (e.g. *transform them into entity types*). Several dozens problem/solution items are proposed. The analyst can select one of them, and execute it automatically or in a controlled way. Alternatively, (s)he can build a script comprising a list of operations, execute it, save and load it.


Global transformations

○ Entity types	into	
Att. entity types	Attributes	
○ Rel-types	into	
Binary N-N	Entity types	
○ Is-a	into	
All	Rel-types	
○ Attributes	into	
Multi-valued	Entity types	
○ Groups	into	
Id. >2 comp.	Add a technical id.	
○ Miscellaneous	into	
Tech. descriptions	Remove	
● Generate	VaxRdb/VMS 4.2	
○ Name processing		

Script

Remove collections
Referential attributes into rel-types
Rel. entity types into rel-types
Attribute entity types into attributes
Remove technical descriptions
Remove all access keys

Figure 12. Control panel of the Transformation assistant. The left-side area is the problem solver, which presents a catalog of problems (1st column) and suggested solutions (2nd column). The right-side area is the script manager. The worksheet shows a simplified script for conceptualizing relational databases.

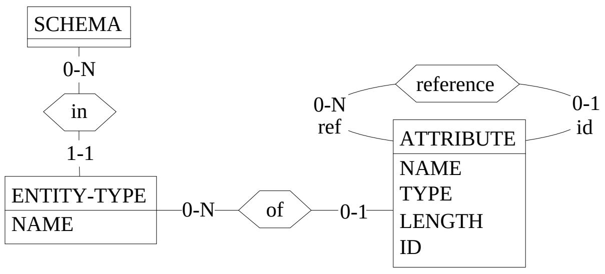
Predefined scripts are available to transform any schema according to popular models (e.g. Bachman model, binary model, relational, CODASYL, standard files), or to perform standard engineering processes (e.g. conceptualization of relational and COBOL schemas, normalization). Customized operations can be added via *Voyager-2* functions (Section 10). Figure 12 shows the control panel of this tool. A second generation of the *Transformation* assistant is under development. It provides a more flexible approach to build complex transformation plans thanks to a catalog of more than 200 preconditions, a library of about 50 actions and more powerful scripting control structures including loops and if-then-else patterns.

The *Schema Analysis* assistant is dedicated to the structural analysis of schemas. It uses the concept of **submodel**, defined as a restriction of the generic specification model described in Section 5 (Hainaut, 92a). This restriction is expressed by a boolean expression of elementary predicates stating which specification patterns are valid, and which ones are forbidden. An elementary predicate can specify situations such as the following : "entity types must have from 1 to 100 attributes", "relationship types have from 2 to 2 roles", "entity type names are less than 18-character long", "names do not include spaces", "no name belongs to a given list of reserved words", "entity types have from 0 to 1 supertype", "the schema is hierarchical", "there are no access keys". A submodel appears as a script which can be saved and loaded. Predefined submodels are available : Normalized ER, Binary ER, NIAM, Functional ER, Bachman, Relational, CODASYL, etc. Customized predicates can be added via *Voyager-2* functions (Section 10). The Schema Analysis assistant offers two functions, namely Check and Search. *Checking* a schema consists in detecting all the constructs which violate the selected submodel, while the *Search* function detects all the constructs which comply with the selected submodel.

The *Text Analysis* assistant presents in an integrated package all the tools dedicated to text analysis. In addition it manages the active links between the source texts and the abstract objects in the repository.

10. Functional Extensibility

DB-MAIN provides a set of built-in standard functions that should be sufficient to satisfy most basic needs in database engineering. However, no CASE tool can meet the requirements of all users in any possible situation, and specialized operators may be needed to deal with unforeseen or marginal situations.



```
function integer MakeForeignKey (string : T1,T2,C1,C2);
explain(*
title="Create a foreign key from an SQL join";
help="if C1 is a unique key of table T1 and if C2 is a column of T2, and if C1 and
C2 are compatible, then define C2 as a foreign key of T2 to T1, and return true,
else return false"
*);
/*define the variables; any repository object type can be a domain */
      schema : S;
      entity_type : E;
      attribute : A,IK,FK;
      list : ID-LIST,FK-LIST;
{
      S := GetCurrentSchema();  /* S is the current schema */
/* ID-LIST = list of the attributes A such that : (1) A belongs to an entity type E which is in schema S and whose name is
T1, (2) the name of A is C1, (3) A is an identifier of E (the ID property of A is true) */
      ID-LIST := attribute[A] {of:entity_type[E]{in:[S] and E.NAME = T1}
                                   and A.NAME = C1
                                   and A.ID = true};
/* FK-LIST = list of the attributes A such that : (1) A belongs to an entity type E which is in S and whose name is T2, (2)
the name of A is C2 */
      FK-LIST := attribute[A] {of:entity_type[E]{in:[S] and E.NAME = T2}
                                   and A.NAME = C2};
/* if both lists are not-empty, then
    if the attributes are compatible then define the attribute in ID-LIST as a foreign key to the attribute in FK-LIST */
      if not(empty(ID-LIST) or empty(FK-LIST)) then
         {IK := GetFirst(ID-LIST); FK := GetFirst(FK-LIST);
          if IK.TYPE = FK.TYPE and IK.LENGTH = FK.LENGTH then
                {connect(reference,IK,FK); return true;}
          else {return false;};}
      else  {return false;};
}
```

Figure 13. A (strongly simplified) excerpt of the repository and a Voyager-2 function which uses it. The repository expresses the fact that schemas have entity types, which in turn have attributes. Some attributes can be identifiers (boolean ID) or can reference (foreign key) another attribute (candidate key). The input arguments of the procedure are four names T1,T2,C1,C2 such as those resulting from an instantiation of the pattern of Figure 11. The function first evaluates the possibility of attribute (i.e. column) C2 of entity type (i.e. table) T2 being a foreign key to entity type T1 with identifier (candidate key) C1. If the evaluation is positive, the referential constraint is created. The `explain` section illustrates the self-documenting facility of Voyager-2 programs; it defines the answers the compiled version of this function will provide when queried by the DB-MAIN tool.

There are two important domains in which users require customized extensions, namely additional internal functions and interfaces with other tools. For instance, analyzing and generating texts in any language and according to any dialect, or importing and exchanging specifications with any CASE tool or Data Dictionary Systems are practically impossible, even with highly parametric import/export processors. To cope with such problems, DB-MAIN provides the *Voyager-2* tool development environment allowing analysts to build their own functions, whatever their complexity. *Voyager-2* offers a powerful language in which specific processors can be developed and integrated into DB-MAIN. Basically, *Voyager-2* is a procedural language which proposes primitives to access and modify the repository through predicative or navigational queries, and to invoke all the basic functions of DB-MAIN. It provides a poweful list manager as well as functions to parse and generate complex text files. A user's tool developed in *Voyager-2* is a program comprising possible recursive procedures and functions. Once compiled, it can be invoked by DB-MAIN just like any basic function.

Figure 13 presents a small but powerful *Voyager-2* function which validates and creates a referential constraint with the arguments extracted from a COBOL/SQL program by the pattern defined in Figure 11. When such a pattern instantiates, the pattern-matching engine passes the values of the four variables T1, T2, C1 and C2 to the *MakeForeignKey* function.

11. Methodological Control and Design Recovery[9]

Though this paper presents it as a CARE tool only, the DB-MAIN environment has a wider scope, i.e. data-centered applications engineering. In particular, it is to address the complex and critical problem of *application evolution*. In this context, understanding how the engineering processes have been carried out when legacy systems were developed, and guiding today's analysts in conducting application development, maintenance and reengineering, are major functions that should be offered by the tool. This research domain, known as *design* (or *software*) *process modeling*, is still under full development, and few results have been made available to practitioners so far. The reverse engineering process is strongly coupled with these aspects in three ways.

First, reverse engineering is an engineering activity of its own (Section 2), and therefore is submitted to rules, techniques and methods, in the same way as forward engineering; it therefore deserves being supported by methodological control functions of the CARE tool.

Secondly, DBRE is a complex process, based on trial-and-error behaviours. Exploring several solutions, comparing them, deriving new solutions from earlier dead-end ones, are common practices. Recording the *history* of a RE project, analyzing it, completing it with new processes, and replaying some of its parts, are typical *design process modeling* objectives.

Thirdly, while the primary aim of reverse engineering is (in short) to recover technical and functional specifications from the operational code of an existing application, a secondary objective is progressively emerging, namely to *recover the design of the application*, i.e. the way the application has (or could have) been developed. This design includes not only the specifications, but also the reasonings, the transformations, the hypotheses and the decisions the development process consists of.

Briefly stated, DB-MAIN proposes a design process model comprising concepts such as *design product, design process, process strategy, decision, hypothesis* and *rationale*. This model derives from proposals such as those of (Potts, 88) and (Rolland, 93), extended to all database engineering activities. This model describes quite adequately not only standard design methodologies, such as

[9] The part of the DB-MAIN project in charge of this aspect is the DB-Process sub-project, fully supported by the Communauté Française de Belgique.

the Conceptual-Logical-Physical approaches (Teorey, 94) (Batini, 92) but also any kind of heuristic design behaviour, including those that occur in reverse engineering. We will shortly describe the elements of this design process model.

Product and product instance. A product instance is any outstanding specification object that can be identified in the course of a specific design. A conceptual schema, an SQL DDL text, a COBOL program, an entity type, a table, a collection of user's views, an evaluation report, can all be considered product instances. Similar product instances are classified into products, such as `Normalized conceptual schema`, `DMS-compliant optimized schema` or `DMS-DDL schema` (see Figure 3).

Process and process instance. A process instance is any logical unit of activity which transforms a product instance into another product instance. Normalizing schema S1 into schema S2 is a process instance. Similar process instances are classified into processes, such as `CONCEPTUAL NORMALIZATION` in Figure 3.

Process strategy. The strategy of a process is the specification of how its goal can be achieved, i.e. how each instance of the process must be carried out. A strategy may be deterministic, in which case it reduces to an algorithm (and can often be implemented as a primitive), or it may be non-deterministic, in which case the exact way in which each of its instances will be carried out is up to the designer. The strategy of a design process is defined by a *script* that specifies, among others, what lower-level processes must/can be triggered, in what order, and under what conditions. The control structures in a script include *action selection* (at most one, one only, at least one, all in any order, all in this order, at least one any number of times, etc.), *alternate actions*, *iteration*, *parallel actions*, *weak condition* (should be satisfied), *strong condition* (must be satisfied), etc.

Decision, hypothesis and rationale. In many cases, the analyst/developer will carry out an instance of a process with some hypothesis in mind. This hypothesis is an essential characteristics of this process instance since it implies the way in which its strategy will be performed. When the engineer needs to try another hypothesis, (s)he can perform another instance of the same process, generating a new instance of the same product. After a while (s)he is facing a collection of instances of this product, from which (s)he wants to choose the best one (according to the requirements that have to be satisfied). A justification of the decision must be provided. Hypothesis and decision justification comprise the design rationale.

History. The history of a process instance is the recorded trace of the way in which its strategy has been carried out, together with the product instances involved and the rationale that has been formulated. Since a project is an instance of the highest level process, its history collects all the design activities, all the product instances and all the rationales that have appeared, and will appear, in the life of the project. The history of a product instance P (also called its *design*) is the set of all the process instances, product instances and rationales which contributed to P. For instance, the design of a database collects all the information needed to describe and explain how the database came to be what it is.

A specific methodology is described in MDL, the DB-MAIN *Methodology Description Language*. The description includes the specification of the products and of the processes the methodology is made up of, as well as of the relationships between them. A product is of a certain type, described as a specialization of a generic specification object from the DB-MAIN model (Section 5), and more precisely as a submodel generated by the *Schema analysis* assistant (Section 9). For instance, a product called `Raw-conceptual-schema` (Figure 3), can be declared as a BINARY-ER-SCHEMA. The latter is a product type that can be defined by a SCHEMA satisfying the following predicate, stating that relationship types are binary, and have no attributes, and that the attributes are atomic and single-valued :

```
(all rel-types have from 2 to 2 roles)
and (all rel-types have from 0 to 0 attributes)
and (all attributes have from 0 to 0 components)
and (all attributes have a max cardinality from 1 to 1);
```

A process is defined mainly by the input product type(s), the internal product type, the output product type(s) and by its strategy.

The DB-MAIN CASE tool is controlled by a **methodology engine** which is able to interpret such a method description once it has been stored in the repository by the MDL compiler. In this way, the tool is customized according to this specific methodology. When developing an application, the analyst carries out process instances according to chosen hypotheses, and builds product instances. (S)he makes decisions which (s)he can justify. All the product instances, process instances, hypotheses, decisions and justifications, related to the engineering of an application make up the trace, or *history* of this application development. This history is also recorded in the repository. It can be examined, replayed, synthesized, and processed (e.g. for design recovery).

One of the most promising applications of histories is database design recovery. Constructing a possible design history for an existing, generally undocumented database is a complex problem which we propose to tackle in the following way. Reverse engineering the database generates a DBRE history. This history can be cleaned by removing unnecessary actions. Reversing each of the actions of this history, then reversing their order, yields a tentative, unstructured, design history. By normalizing the latter, and by structuring it according to a reference methodology, we can obtain a possible design history of the database. Replaying this history against the recovered conceptual schema should produce a physical schema which is equivalent to the current database.

A more comprehensive description of how these problems are addressed in the DB-MAIN approach and CASE tool can be found in (Hainaut, 94), while the design recovery approach is described in (Hainaut, 96b).

12. DBRE Requirements and the DB-MAIN CASE Tool

We will examine the requirements described in Section 3 to evaluate how the DB-MAIN CASE tool can help satisfy them.

Flexibility : instead of being constrained by rigid methodological frameworks, the analyst is provided with a collection of neutral toolsets that can be used to process any schema whatever its level of abstraction and its degree of completion. In particular, backtracking and multi-hypothesis exploration are easily performed. However, by customizing the method engine, the analyst can build a specialized CASE tool that is to enforce strict methodologies, such as that which has been described in Section 2.

Extensibility : through the *Voyager-2* language, the analyst can quickly develop specific functions; in addition, the assistants, the name and the text analysis processors allows the analyst to develop customized scripts.

Sources multiplicity : the most common information sources have a text format, and can be queried and analyzed through the text analysis assistant. Other sources can be processed through specific *Voyager-2* functions. For example, data analysis is most often performed by small ad hoc queries or application programs, which validate specific hypotheses about, e.g., a possible identifier or foreign key. Such queries and programs can be generated by *Voyager-2* programs that implement heuristics about the discovery of such concepts. In addition, external information processors and analyzers can easily introduce specifications through the text-based import-export ISL language.

For example, a simple SQL program can extract SQL specifications from DBMS data dictionaries, and generate their ISL expression, which can then be imported into the repository.

Text analysis : the DB-MAIN tool offers both general purpose and specific text analyzers and processors. If needed, other processors can be developed in *Voyager-2*. Finally, external analyzers and text processors can be used provided they can generate ISL specifications which can then be imported in DB-MAIN to update the repository.

Name processing : besides the name processor, specific *Voyager-2* functions can be developed to cope with more specific name patterns or heuristics. Finally, the compact and sorted views can be used as poweful browsing tools to examine name patterns or to detect similarities.

Links with other CASE processes : DB-MAIN is not dedicated to DBRE only; therefore it includes in a seamless way supporting functions for the other DB engineering processes, such as forward engineering. Being neutral, many functions are common to all the engineering processes.

Openness : DB-MAIN supports exchanges with other CASE tools in two ways. First, Voyager-2 programs can be developed (1) to generate specifications in the input language of the other tools, and (2) to load into the repository the specifications produced by these tools. Secondly, ISL specifications can be used as a neutral intermediate language to communicate with other processors.

Flexible specification model : the DB-MAIN repository can accomodate specifications of any abstraction level, and based on a various paradigms; if asked to be so, DB-MAIN can be fairly tolerant to incomplete and inconsistent specifications and can represent schemas which include objects of different levels and of different paradigms (see Figure 5); at the end of a complex process the analyst can ask, through the Schema Analysis assistant, a precise analysis of the schema to sort out all the structural flaws.

Genericity : both the repository schema and the functions of the tool are independent of the DMS and of the programming languages used in the application to be analyzed. They can be used to model and to process specifications initially expressed in various technologies. DB-MAIN includes several ways to specialize the generic features in order to make them compliant with a specific context, such as processing PL/1-IMS, COBOL-VSAM or C-ORACLE applications.

Multiplicity of views : the tool proposes a rich palette of presentation layouts both in graphical and textual formats. In the next version, the analyst will be allowed to define customized views.

Rich transformation toolset : DB-MAIN proposes a transformational toolset of more than 25 basic functions; in addition, other, possibly more complex, transformations can be built by the analyst through specific scripts, or through *Voyager-2* functions.

Traceability : DB-MAIN explicitly records a *history*, which includes the successive states of the specifications as well as all the engineering activities performed by the analyst and by the tool itself. Viewing these activities as specification transformations has proved an elegant way to formalize the links between the specifications states. In particular, these links can be processed to explain how a conceptual object has been implemented (forward mapping), and how a technical object has been interpreted (reverse mapping).

13. Implementation and Applications of DB-MAIN

We have developed DB-MAIN in C++ for MS-Windows machines. The repository has been implemented as an object oriented database. For performance reasons, we have built a specific OO database manager which provides very short access and update times, and whose disc and core memory requirements are kept very low. For instance, a fully documented 40,000-object project can be developed on a 8-MB machine.

The first version of DB-MAIN was released in September 1995. It includes the basic processors and functions required to design, implement and reverse engineer large size databases according to various DMS. Version 1 supports many of the features that have been described in this paper. Its repository can accomodate data structure specifications at any abstraction level (Section 5). It provides a 25-transformation toolkit (Section 6), four textual and two graphical views (Section 7), parsers for SQL, COBOL, CODASYL, IMS and RPG programs, the PDL pattern-matching engine, the dataflow graph inspector, the name processor (Section 8), the Transformation, Schema Analysis and Text Analysis assistants (Section 9), the *Voyager-2* virtual machine and compiler (Section 10), a simple history generator and its replay processor (Section 11). Among the other functions of Version 1, let us mention code generators for various DMS. Its estimated cost was about 20 man/year.

The DB-MAIN tool has been used to carry out several government and industrial projects. Let us describe five of them briefly.

• *Design of a government agricultural accounting system.*

The initial information was found in the notebooks in which the farmers record the day-to-day basic data. These documents were manually encoded as giant entity types with more than 1850 attributes and up to 9 decomposition levels. Through conceptualization techniques, these structures were transformed into pure conceptual schemas of about 90 entity types each. Despite the unusual context for DBRE, we have followed the general methodology described in Section 2 :

- *Data structure extraction* : manual encoding; refinement through direct contacts with selected accounting officers;
- *Data structure conceptualization :*
  - *Untranslation* : the multivalued and compound attributes have been transformed into entity types; the entity types with identical semantics have been merged; serial attributes, i.e. attributes with similar names and identical types, have been replaced with multivalued attributes;
  - *De-optimization* : the farmer is requested to enter the same data at different places; these redundancies have been detected and removed; the calculated data have been removed as well;
  - *Normalization* : the schema included several implicit IS-A hierarchies, which have been expressed explicitly;

The cost for encoding, conceptualizing and integrating three notebooks was about 1 person/month. This rather unusual application of reverse engineering techiques was a very interesting experience because it proved that data structure engineering is a global domain which is difficult (and sterile) to partition into independent processes (design, reverse). It also proved that there is a strong need for highly generic CASE tools.

• *Migrating a hybrid file/SQL social security system into a pure SQL database.*

Due to a strict disciplined design, the programs were based on rather neat file structures, and used systematic *clichés* for integrity constraints management. This fairly standard two-month project comprised an interesting work on name patterns to discover foreign keys. In addition, the file structures included complex identifying schemes which were difficult to represent in the DB-MAIN repository, and which required manual processing.

• *Redocumenting the ORACLE repository of an existing OO CASE tool.*

Starting from various SQL scripts, partial schemas were extracted, then integrated. The conceptualization process was fairly easy due to systematic naming conventions for candidate and foreign keys. In addition, it was performed by a developer having a deep knowledge of the database. The process was completed in two days.

• *Redocumentating a medium size ORACLE hospital database.*

The database included about 200 tables and 2,700 columns. The largest table had 75 columns. The analyst quickly detected a dozen major tables with which one hundred views were associated. It appeared that these views defined, in a systematic way, a 5-level subtypes hierarchy. Entering the description of these subtypes by hand would have required an estimated one week. We chose to build a customized function in *PDL* and *Voyager-2* as follows. A pattern was developed to detect and analyze the `create view` statements based on the main tables. Each instantiation of this pattern triggered a *Voyager-2* function which defined a subtype with the extracted attributes. Then, the function scanned these IS-A relations, detected the common attributes, and cleaned the supertype, removing inherited attributes, and leaving the common ones only. This tool was developed in 2 days, and its execution took 1 minute. However, a less expert *Voyager-2* programmer could have spent more time, so that these figures cannot be generalized reliably. The total reverse engineering process cost 2 weeks.

• *Reverse engineering of an RPG database.*

The application was made of 31 flat files comprising 550 fields (2 to 100 fields per file), and 24 programs totalling 30,000 LOC. The reverse engineering process resulted in a conceptual schema comprising 90 entity types, including 60 subtypes, and 74 relationship types. In the programs, data validation concentrated in well defined sections. In addition, the programs exhibited complex access patterns. Obviously, the procedural code was a rich source of hidden structures and constraints. Due to the good quality of this code, the program analysis tools were of little help, except to quickly locate some statements. In particular, pattern detection could be done visually, and program slicing yielded too large program chunks. Only the dataflow inspector was found useful, though in some programs, this graph was too large, due to the presence of working variables common to several independent program sections. At that time, no RPG parser was available, so that a *Voyager-2* RPG extractor was developed in about one week. The final conceptual schema was obtained in 3 weeks. The source file structures were found rather complex. Indeed, some non-trivial patterns were largely used, such as overlapping foreign keys, conditional foreign and primary keys, overloaded fields, redundancies (Blaha, 95). Surprisingly, the result was estimated unnecessarily complex as well, due to the deep type/subtype hierarchy. This hierarchy was reduced until it seemed more tractable. This problem triggered an interesting discussion about the limit of this inheritance mechanism. It appeared that the precision *vs* readability trade-off may lead to unnormalized conceptual schemas, a conclusion which was often formulated against object class hierarchies in OO databases, or in OO applications.

14. Conclusions

Considering the requirements outlined in Section 3, few (if any) commercial CASE/CARE tools offer the functions necessary to carry out DBRE of large and complex applications in a really effective way. In particular, two important weaknesses should be pointed out. Both derive from the oversimplistic hypotheses about the way the application was developed. First, extracting the data structures from the operational code is most often limited to the analysis of the data structure declaration statements. No help is provided for further analyzing, e.g., the procedural sections of the programs, in which essential additional information can be found. Secondly, the logical schema is considered as a straighforward conversion of the conceptual schema, according to simple translating rules such as those found in most textbooks and CASE tools. Consequently, the conceptualization phase uses simple rules as well. Most actual database structures appear more sophisticated, however, resulting from the application of non standard translation rules and including sophisticated performance oriented constructs. Current CARE tools are completely blind to such structures, which they carefully transmit into the conceptual schema, producing, e.g., *optimized IMS conceptual schemas*, instead of pure conceptual schemas.

The DB-MAIN CASE tool presented in this paper includes several CARE components which try to meet the requirements described in Section 3. The first version[10] has been used successfully in several real size projects. These experiments have also put forward several technical and methodological problems, which we describe briefly.

- *Functional limits of the tool*. Though DB-MAIN Version 1 already offers a reasonable set of integrity constraints, a more powerful model was often needed to better describe physical data structures or to express semantic structures. Some useful schema transformations were lacking, and the scripting facilities of the assistants were found very interesting, but not powerful enough in some situations. As expected, several users asked for "*full program reverse engineering*".

- *Problem and tool complexity*. Reverse engineering is a software engineering domain based on specific, and still unstable, concepts and techniques, and in which much remains to learn. Not surprisingly, true CARE tools are complex, and DB-MAIN is no exception when used at its full potential. Mastering some of its functions requires intensive training which can be justified for complex projects only. In addition, writing and testing specific *PDL* pattern libraries and *Voyager-2* functions can cost several weeks.

- *Performance*. While some components of DB-MAIN proved very efficient when processing large projects with multiple sources, some others slowed down as the size of the specifications grew. That was the case when the pattern-matching engine parsed large texts for a dozen patterns, and for the dataflow graph constructor which uses the former. However, no dramatic improvement can be expected, due to the intrinsic complexity of pattern-matching algorithms for standard machine architectures.

- *Viewing the specifications*. When a source text has been parsed, DB-MAIN builds a first-cut logical schema. Though the tool proposes automatic graphical layouts, positioning the extracted objects in a natural way is up to the analyst. This task was often considered painful, even on a large screen, for schemas comprising a many objects and connections. In the same realm, several users

[10] In order to develop contacts and collaboration, an *Education* version (complete but limited to small applications) and its documentation have been made available. This free version can be obtained by contacting the first author at `jlh@info.fundp.ac.be`.

found that the graphical representations were not as attractive as expected for very large schemas, and that the textual views often proved more powerful and less cumbersome.

The second version, which is under development, will address several of the observed weaknesses of Version 1, and will include a richer specification model and extended toolsets. We will mainly mention some important extensions : a view derivation mechanism, which will solve the problem of mastering large schemas, a view integration processor to build a global schema from extracted partial views, the first version of the MDL compiler, of the methodology engine, and of the history manager, and an extended program slicer. The repository will be extended to the representation of additional integrity constraints, and of other system components such as programs. A more powerful version of the *Voyager-2* language and a more sophisticated *Transformation* assistant (evoked in Section 9) are planned for Version 2 as well. We also plan to experiment the concept of design recovery for actual applications.

Acknowledgements

The detailed comments by the anonymous reviewers have been most useful to improve the readability and the consistency of this paper, and to make it as informative as possible. We also like to thank Linda Wills for her friendly encouragements.

References

Andersson, M. 1994. Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

Batini, C., Ceri, S., Navathe, S., B. 1992. *Conceptual Database Design,* Benjamin/ Cummings

Batini, C., Di Battista, G., Santucci, G. 1993. Structuring Primitives for a Dictionary of Entity Relationship Data Schemas, *IEEE TSE,* Vol. 19, No. 4

Blaha, M.R., Premerlani, W., J. 1995. Observed Idiosyncracies of Relational Database designs, in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering,* Toronto, July 1995, IEEE Computer Society Press

Bolois, G., Robillard, P. 1994. Transformations in Reengineering Techniques, in *Proc. of the 4th Reengineering Forum "Reengineering in Practice",* Victoria, Canada

Casanova, M., Amarel de Sa, J. 1983. Designing Entity Relationship Schemas for Conventional Information Systems, in *Proc. of ERA,* pp. 265-278

Casanova, M., A., Amaral De Sa. 1984. Mapping uninterpreted Schemes into Entity-Relationship diagrams : two applications to conceptual schema design, in *IBM J. Res. & Develop*., Vol. 28, No 1

Chiang, R., H., Barron, T., M., Storey, V., C. 1994. Reverse Engineering of Relational Databases : Extraction of an EER model from a relational database, *Journ. of Data and Knowledge Engineering*, Vol. 12, No. 2 (March 94), pp107-142

Date, C., J. 1994. *An Introduction to Database Systems*, Volume 1, Addison-Wesley

Davis, K., H., Arora, A., K. 1985. A Methodology for Translating a Conventional File System into an Entity-Relationship Model, in *Proc. of ERA,* IEEE/North-Holland

Davis, K., H., Arora, A., K. 1988. Converting a Relational Database model to an Entity Relationship Model, in *Proc. of ERA : a Bridge to the User*, North-Holland

Edwards, H., M., Munro, M. 1995. Deriving a Logical Model for a System Using Recast Method, in *Proc. of the 2nd IEEE WC on Reverse Engineering*, Toronto, IEEE Computer Society Press

Fikas, S., F. 1985. Automating the transformational development of software, *IEEE TSE*, Vol. SE-11, pp1268-1277

Fong, J., Ho, M. 1994. Knowledge-based Approach for Abstracting Hierarchical and Network Schema Semantics, in *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, Springer-Verlag

Fonkam, M., M., Gray, W., A. 1992. An approach to Eliciting the Semantics of Relational Databases, in *Proc. of 4th Int. Conf. on Advance Information Systems Engineering* - CAiSE'92, pp. 463-480, May, LNCS, Springer-Verlag

Elmasri, R., Navathe, S. 1994. *Fundamentals of Database Systems*, Benjamin-Cummings

Hainaut, J.-L. 1981. Theoretical and practical tools for data base design, in *Proc. Intern. VLDB conf.*, ACM/IEEE

Hainaut, J-L. 1991a. Entity-generating Schema Transformation for Entity-Relationship Models, in *Proc. of the 10th ERA*, San Mateo (CA), North-Holland

Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O. 1992. Database CASE Tool Architecture : Principles for Flexible Design Strategies, in *Proc. of the 4th Int. Conf. on Advanced Information System Engineering* (CAiSE-92), Manchester, May 1992, Springer-Verlag, LNCS

Hainaut, J-L., Chandelon M., Tonneau C., Joris M. 1993. 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, J-L, Chandelon M., Tonneau C., Joris M. 1993b. Transformational techniques for database reverse engineering, in *Proc. of the 12th Int. Conf. on ER Approach, Arlington-Dallas*, E/R Institute and Springer-Verlag, LNCS

Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D. 1994. Evolution of database Applications : the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

Hainaut, J-L. 1995. *Transformation-based database engineering*,Tutorial notes, VLDB'95, Zürich, Switzerland, Sept. 1995 (available at jlh@info.fundp.ac.be)

Hainaut, J-L. 1996. Specification Preservation in Schema transformations - Application to Semantics and Statistics, *Data & Knowledge Engineering*, Elsevier (to appear)

Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. 1996b. *Database design recovery,* DB-MAIN Research Paper, (available at jlh@info.fundp.ac.be)

Halpin, T., A., Proper, H., A. 1995. Database schema transformation and optimization, in *Proc. of the 14th Int. Conf. on ER/OO Modelling (ERA)*, Springer-Verlag

Hall, P., A., V. (Ed.) 1992. *Software Reuse and Reverse Engineering in Practice*, Chapman&Hall

IEEE, 1990. *Special issue on Reverse Engineering*, IEEE Software, January, 1990

Johannesson, P., Kalman, K. 1990. *A* Method for Translating Relational Schemas into Conceptual Schemas, in *Proc. of the 8th ERA*, Toronto, North-Holland,

Joris, M., Van Hoe, R., Hainaut, J-L., Chandelon M., Tonneau C., Bodart F. et al. 1992. *PHENIX : methods and tools for database reverse engineering*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, December 1992, EC2 Publish.

Kobayashi, I. 1986. Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence, in *Information Systems*, Vol. 11, No 1, pp. 41-59

Kozaczynsky, Lilien, 1987. An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation, in *Proc. of ERA Conf.*

Markowitz, K., M., Makowsky, J., A. 1990. Identifying Extended Entity-Relationship Object Structures in Relational Schemas, *IEEE Trans. on Software Engineering*, Vol. 16, No. 8

Navathe, S., B. 1980. Schema Analysis for Database Restructuring, in *ACM TODS*, Vol.5, No.2

Navathe, S., B., Awong, A. 1988. Abstracting Relational and Hierarchical Data with a Semantic Data Model, in *Proc. of ERA : a Bridge to the User*, North-Holland

Nilsson,E., G. 1985. The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema, in *Proc. of ERA*, IEEE/North-Holland,

Petit, J-M., Kouloumdjian, J., Bouliaut, J-F., Toumani, F. 1994. Using Queries to Improve Database Reverse Engineering, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

Premerlani, W., J., Blaha, M.R. 1993. An Approach for Reverse Engineering of Relational Databases, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press

Potts, C., Bruns, G. 1988. Recording the Reasons for Design Decisions, in *Proc. of ICSE*, IEEE Computer Society Press

Rauh, O., Stickel, E. 1995. Standard Transformations for the Normalization of ER Schemata, in *Proc. of the CAiSE•95 Conf.*, Jyväskylä, Finland, LNCS, Springer-Verlag

Rock-Evans, R. 1990. *Reverse Engineering : Markets, Methods and Tools*, OVUM report

Rosenthal, A., Reiner, D. 1988. Theoretically sound transformations for practical database design, in *Proc. of ERA Conf.*

Rosenthal, A., Reiner, D. 1994. Tools and Transformations - Rigourous and Otherwise - for Practical Database Design, *ACM TODS*, Vol. 19, No. 2

Rolland, C. 1993. Modeling the Requirements Engineering Process, in *Proc of the 3rd European-Japanese Seminar in Information Modeling and Knowledge Bases*, May 1993, Budapest (preprints)

Sabanis, N., Stevenson, N. 1992. Tools and Techniques for Data Remodelling Cobol Applications, in *Proc. 5th Int. Conf. on Software Engineering and Applications*, Toulouse, 7-11 December, pp. 517-529, EC2 Publish.

Selfridge, P., G., Waters, R., C., Chikofsky, E., J. 1993. Challenges to the Field of Reverse Engineering, in *Proc. of the 1st WC on Reverse Engineering*, pp.144-150, IEEE Computer Society Press

Shoval, P., Shreiber, N. 1993. Database Reverse Engineering : from Relational to the Binary Relationship Model, *Data and Knowledge Engineering*, Vol. 10, No. 10

Signore, O, Loffredo, M., Gregori, M., Cima, M. 1994. Reconstruction of ER Schema from Database Applications: a Cognitive Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag

Springsteel, F., N., Kou, C. 1990. Reverse Data Engineering of E-R designed Relational schemas, in *Proc. of Databases, Parallel Architectures and their Applications*

Teorey, T. J. 1994. *Database Modeling and Design : the Fundamental Principles*, Morgan Kaufmann

Vermeer, M., Apers, P. 1995. Reverse Engineering of Relational Databases, in *Proc. of the 14th Int. Conf. on ER/OO Modelling (ERA*)

Weiser, M. 1984. Program Slicing, *IEEE TSE*, Vol. 10, pp 352-357

Wills, L., Newcomb, P., Chikofsky, E., (Eds) 1995. *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press

Winans, J., Davis, K., H. 1990. Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model, in *Proc. of ERA : the Core of Conceptual Modelling*, pp. 345-360, October, North-Holland