Requirements for Information System Reverse Engineering Support

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

Abstract

This paper proposes a general architecture for Information systems (or data-centered applications) reverse engineering CASE environments. Recovering the specifications of such applications requires recovering first those of their data, i.e. database reverse engineering (DBRE). First, the paper describes a generic DMS-independent DBRE methodology, then it analyses the main characteristics of DBRE activities in order to collect a set of minimum or desired requirements. Finally, it describes the main features of an operational CASE tool developed according to these requirements. This study and these developments are being carried out as part of the DB-MAIN and DB-PROCESS projects[1]

1. INTRODUCTION

Reverse engineering (RE) 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 [I1,H7]. Recovering these specifications is generally intended to redocument, convert, restructure, maintain or extend old applications. The problem is particularly complex with old and ill-designed applications. In this case, not only no decent documentation (if any) can be relied on, but the lack of systematic methodologies for designing and maintaining them have led to tricky and obscure code.

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 :

- 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 structure* is 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. Though RE data structures still is a complex task, it appears that the current state of the art provides us with sufficiently powerful concepts and techniques to make this enterprise more realistic. The literature proposes systematic approaches for database schema recovering : standard files [N3,S1], IMS [N2,B1], CODASYL [B1], relational databases [A1,B1,C3, D3,F3,J1,M1,N2,P1,P2,S3,S4,S5].

Most of these studies, however, appear to be limited in scope, and are generally based on assumptions on the quality and completeness of the source data structures to reverse engineer 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),
- the translation is rather straightforward (no tricky representations); for instance, a relational schema often is supposed to be in 4NF; [P2] is one of the only proposals that cope with some non trivial representations;
- the schema has not been deeply restructured for performance objectives or for any other requirements,
- a complete 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).

In many proposals, it appears that the only databases processable are those which 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 1993, some authors recognize that

[1] The DB-MAIN project is partially supported by ACEC-OSI, ARIANE-II, Banque UCL (Lux), Centre de recherche public H. Tudor (Lux), CGER, Cockrill-Sambre, CONCIS (Fr), D'Ieteren, DIGITAL, EDF (Fr), Groupe S, IBM, OBLOG Software (Port), ORIGIN, Winterthur, 3 Suisses. It is developed in collaboration with the Database Lab. of the EPFL (Lausanne, CH). The DB-PROCESS subproject is supported by the *Communauté Française de Belgique*.

the procedural part of the application programs is an essential source of information on the data structures [J2,H4,P1,A1, S4]. Being a complex process, DBRE cannot be successful without the support of adequate tools. An increasing number of commercial products (claim to) offer DBRE functionalit-ies. Though they ignore many of the most difficult aspects of the problem, these tools provide their users with invaluable help to carry out DBRE more effectively.

In [H4], we proposed the theoretical baselines for a generic, DBMS-independent, DBRE methodology. These baselines have been developed and extended in [H5] and [H6]. The current paper translates these principles into practical requirements DBRE CASE tools should satisfy, and presents the main aspects and components of a prototype CASE tool dedicated to database applications engineering, and more specifically to database reverse engineering.

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 CASE 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), Text analysers and Name processor (section 8), the Assistants (section 9), Functional extensibility (section 10) and Methodological control (section 11).

2. A GENERIC METHODOLOGY FOR DATABASE REVERSE ENGINEERING

The problems that arise when recovering the documentation of the data naturally fall in two categories that make the two major processes in DBRE, namely *data structure extraction* and *data structure conceptualization* ([J2,H4]). 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 [T1,B1]). 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. The methodology, as developed in [H4], is sketched in Fig. 1.

2.1 Data structure extraction

This phase consists in recovering the complete DMS[2] 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, etc).

The problem is much more complex for standard files, for which no computerized description of their structure exists in most cases[3]. Each source program must be analysed in order to detect partial structures of the files. For most real-world (i.e. non academic) applications, this analysis goes well beyond the mere detection of 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 also are common in (true) databases, i.e. those controlled by DBMS.

Structure hiding applies on a source data structure or constraint S1, *which could be implemented in the target 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. For instance, 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 applications. Most often, the checking sections are not centralized, but are distributed and duplicated (frequently in different versions), throughout the application programs. Some examples : referential integrity in standard files, 1-1 relationship types in CODASYL DB.

Lost specifications are constructs of the conceptual schema which 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

---

2 A Data Management System (DMS) is either a File Management System (FMS) or a DataBase Management System (DBMS).

3 Though some practices (disciplined use of COPY or INCLUDE meta statements), and some tools (data dictionaries) may simulate such centralized schemas.

constraint[4], but the trace of its enforcement can not be found in the declared data structures nor in the application programs. Some popular examples : uniqueness constraints on sequential files, secondary identifiers 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 HM interfaces, of the organizational rules, can accumulate evidences for these specifications. Most often, these evidences 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 [P1,A1,S4].

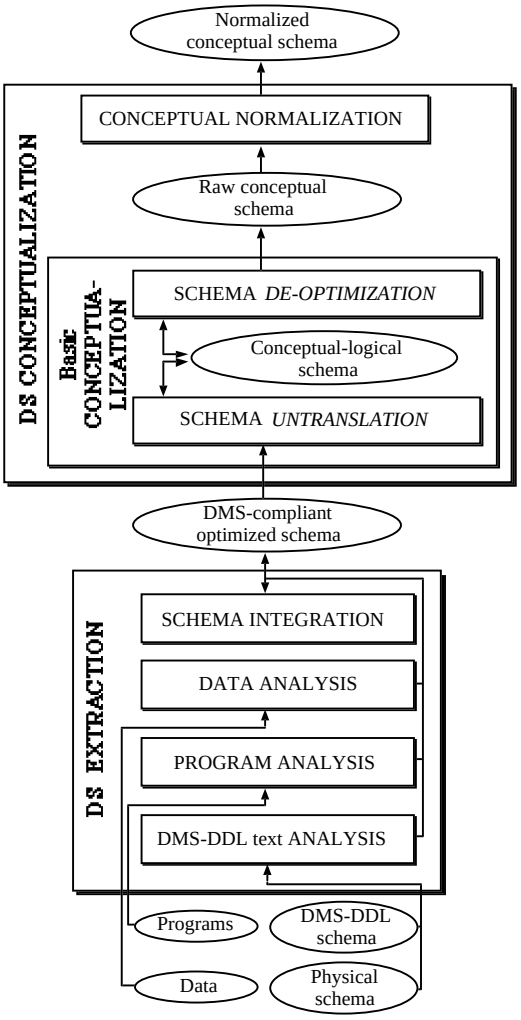


Figure 1 - Main components of the generic DBRE methodology.

The main processes of DATA STRUCTURE EXTRACTION are the following :

DMS-DDL text ANALYSIS

This rather straighforward process consists in analysing 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 analysing the other parts of the application programs, a.o. the procedural sections, in order to detect evidences 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 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 (in fact *may*) be found in this way.

SCHEMA INTEGRATION

When more than one source have 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 CASE 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.

2.2 Data structure conceptualisation

This second phase addresses the conceptual interpretation of the DBMS schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DMS-dependent constructs. It consists in two sub-processes, namely *Basic conceptualization* and *Conceptual normalization*. The reader will find in [H5] a more detailed development of these processes, which heavily rely on schema restructuring techniques (or schema transformations).

4 No miracle here : for instance, the data are imported, or organizational rules make them satisfy these constraints.

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*.

• *Schema untranslation* : the logical schema is the technical translation of conceptual constructs into the DMS model. Through this process, the analyst identifies the traces of such translations, and replaces them by their origin 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 important subsets 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).

• *Schema de-optimization* : the logical schema is searched for evidences of constructs designed for optimization purposes and these constructs are discarded [H5].

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 [B1,T1].

All proposals address this phase, generally for specific DMS, and for rather simple schemas (e.g. with no implementation tricks). They generally propose rules and heuristics for the *SCHEMA UNTRANSLATION* process.

2.3 Conclusion

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.

Consequently this methodological framework can be specialized according to a specific DMS and according to specific development standards. For instance [H5] suggests specialized versions of the Conceptualization phase for SQL, COBOL, IMS, CODASYL and TOTAL/IMAGE.

3. REQUIREMENTS FOR A CARE[5] TOOL

This section tries to state some of the most important requirements an ideal DBRE support environment (or CARE tool) should satisfy. These requirements are induced by the analysis of the specific characteristics of DBRE processes.

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, basically is 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 very flexible working patterns, included unstructured ones. It should be methodology-neutral as opposed to forward engineering tools. A toolbox architecture is recommended. In addition, the tool must be be highly interactive.

Observation : RE appears as a learning process : RE projects often are new problems of their own, requiring specific reasonings and techniques.

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

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, 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.

Observation : More particularly, database RE requires browsing through huge amounts of text, searching them for specific patterns (e.g. programming *clichés*), following static execution paths and dataflows, extracting program slices [W1].

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

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[6], 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.

Observation : RE is seldom an independent activity. For instance, (1) forward engineering projects frequently include reverse engineering of some existing components,

5 For Computer-Aided Reverse Engineering.

6 For instance, Belgium uses three legal languages, namely French, Dutch and German. English is often used as *de facto* common language.

(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 include a large set of functions, a.o. those which pertain to forward engineering.

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 several CASE tools, DBMS, 4GL or DDS.

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

Observation : As in any CAD activity, RE applies on incomplete and inconsistent specifications. 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 allow the representation of components of different abstraction levels.

Observation : Tricks and implementation techniques specific of some data models have been found to be used in the other data models as well (e.g. foreign keys 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.

Observation : The specifications, whatever their abstraction level (e.g. physical, logical or conceptual), most often are huge and complex, and need being 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.

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 ([H2,H5]).

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.

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.

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 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 (included full control on logical and physical processes), database reverse engineering, database application reengineering, maintenance, migration and evolution. Further detail can be found in [H6].

As far as DBRE support is concerned, the DB-MAIN CASE tool tries to address the requirements developed in section 3. As a large-scope CASE tool, DB-MAIN includes the usual functions needed in DB analysis and design, e.g. entry, browsing, management, validation and transformation of specifications, as well as code and report generation. The following sections will concentrate on the main aspects and components of the tool which are directly related to DBRE activities, namely :

- the data structure representation model, and the repository (section 5),
- the transformational approach and the transformation toolboxes (section 6),
- the user interface (section 7),
- text analysers and name processor (section 8),
- the specialized assistants (section 9),
- functional extensibility (section 10)
- methodological control and guidance (section 11).

5. THE DB-MAIN SPECIFICATION MODEL AND REPOSITORY

The repository collects all the information related to a project. When we consider the database aspects only (though they have strong links with the data structures in DBRE, the specification of the other aspects of the application, e.g. processing, will be ignored in this paper), the repository comprises three classes of information :

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

We will ignore the two latter classes, related to methodological control. They will be evoked briefly in section 11, and have been discussed in [H6].

A *schema* is a description of the data structures to be processed, while a *text* is any textual material generated or analysed 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 paper.

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 [H3]:

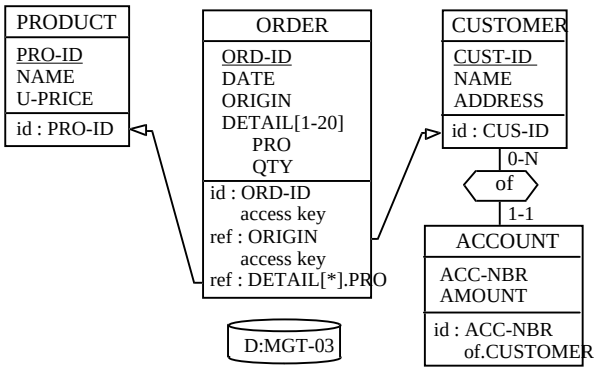


Figure 2 - A typical data structure schema during reverse engineering. This schema includes conceptualized objects (PRODUCT, CUSTOMER, ACCOUNT, of), logical objects (record type ORDER, with single-valued and multivalued foreign keys) and physical objects (access keys ORDER.ORD-ID and ORDER.ORIGIN; file D:MGT-03)

conceptual constructs : entity types (with/without attributes; with/without identifiers), super/subtype hierarchies (single/multiple inheritance, total and disjoint properties), relationship types (binary/N-ary; cyclic/acyclic), roles of relationship type (with min-max cardinalities; with/without explicit name; single/multi-entity-type), attributes (of entity and relationship types; multi/single-valued; atomic/compound), identifiers (of entity type, relationship type, attribute; comprising attributes and/or roles), constraints (inclusion, exclusion, coexistence, etc);

logical constructs : record types, fields, referential constraints, redundancy, etc;

physical constructs : entity collections (abstracting files, datasets, table spaces, etc), access keys (abstracting 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 level of abstraction. Figure 2 illustrates this fact through a schema which includes conceptual, logical and physical constructs. Ultimately, this schema will be completely conceptualized through processing of the logical and physical constructs.

6.THE TRANSFORMATION TOOLKIT

The desirability of the transformational approach to software engineering is now widely recognized. According to [F1] for instance, *the process of developing a program [can be] formalized as a set of transformations*. Since several years, this approach is put forward in database engineering by an increasing number of authors, either in research papers, or in text books and, more recently, in several CASE tools ([H3,R3]). Quite naturally, schema transformations have found their way in DBRE as well [H4,H5,B3]. The transformational approach is the cornerstone of the DB-MAIN approach [H1,H2,H5,H6] and CASE tool [H3,J2, H6]. A formal presentation can be found in [H2].

Grossly speaking, a schema transformation consists in deriving a target schema S' from 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 [N1,K1,K2, R2,B1]. Moreover, some authors claim that the whole database design process, together with other related activities, can be described as a chain of schema transformations [B2,H5,R3]. Schema transformations are essential to formally define forward et backward mappings between schemas, and particularly betwen conceptual structures and DMS constructs. Fig. 3, 4 and 5 show some popular transformations.

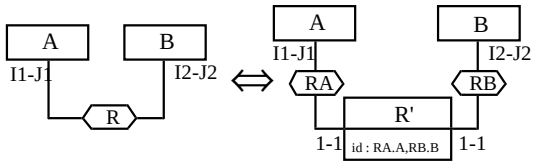


Fig. 3 - Transforming a binary relationship type into an entity type, and conversely.

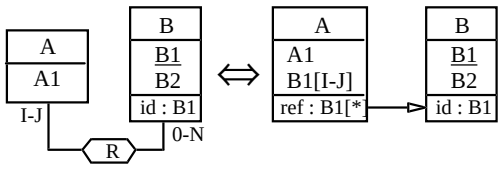


Fig. 4 - Relationship-type R is represented by foreign key B1, and conversely. If J > 1, the foreign key is multivalued.

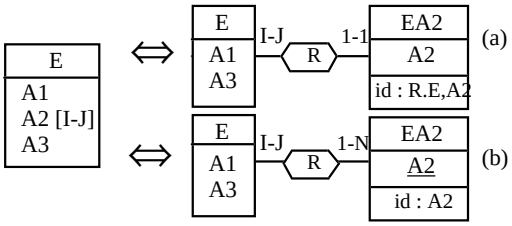


Fig. 5 - Transformation of an attribute into an entity type : (a) by explicit representation of its **instances**, (b) by explicit representation of its distinct **values** (and conversely).

DB-MAIN proposes a three-level transformation toolset. These tools are generic, in that they can be used in any database engineering processes (see [H3] for more detail).

• **elementary transformations** : a transformation is applied to one object; with these tools, the user keeps full control on the schema transformation since similar situat-ions can be solved by different transformations; e.g. a multivalued attribute can be transformed in a dozen of ways;

• **global transformations** : a transformation is applied to all the relevant objects of a schema. This tool-set is controlled by the Transformation Asistant (section 8). Examples : replace all one-to-many relationship types by foreign keys; replace all multivalued attributes by entity types + many-to-one relationship types.

• **model-driven transformations** : all the constructs of a schema that do not comply with a given model are transformed; these transformations require little control from the user; the resulting schema is correct, complies with, say, the relational or CODASYL model, but has few refinements as far as efficiency is concerned. The analyst can build its own model-driven transformations through V2 procedures (section 10), and through scripting facilities of the Transformation assistant (section 9).

7. THE USER INTERFACE

The user interaction is through a fairly standard GUI. However, interacting with the specifications exhibits some original options which deserves being mentioned.

Browsing through, processing, and analysing, large schemas require 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 to analyse name correspondances and similarities. DB-MAIN currently offers six ways of presenting a schema (four hypertext views and two graphical views).

8. TEXT ANALYSIS and NAME PROCESSING

Text analysis will occur mainly in two specific processes, namely `DMS-DDL text ANALYSIS` and `PROGRAM ANALYSIS`.

The first process is rather simple, and can be carried out by **extractors**. Though DB-MAIN currently offers a limited set of built-in standard parsers (COBOL, SQL, CODASYL, IMS), the ultimate goal is to let analysts customize generic parsers through V2 functions (section 10).

To address the requirements of the second process, through which the preliminary specifications are refined, DB-MAIN includes an interactive **pattern-matching engine** which can search large text files for definite patterns or *clichés* expressed in PDL, a *Pattern Definition Language* whose kernel is close to BNF notation and to UNIX *grep*.

Fig. 6 illustrates one of the popular heuristics to detect an implicit foreign key in a relational schema [S4,A1,P1]. The principle is simple : most multitable queries use primary/foreign key joins. Therefore, any SQL expression that looks like "`select..from..A,..B..where..A.A1=B.B1..`" (and some other forms) suggests, provided A1 is a key of A, that B1 is a foreign key of A to B. This is just what Fig. 6 translates in a formal way. This example exhibits two essential features of PDL and of its engine.

The SQL generic patterns

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

The COBOL/DB2 specific patterns

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

Figure 6 - 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 by 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.

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, valid for any RDBMS and any host language, while the specific

patterns complete 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 (prefixed with @). When such a pattern is instantiated, the variables are given a value which can be used, e.g., to update the repository.

The pattern engine can analyse external source files, as well as textual descriptions stored in the repository (where, for instance, the extractors store the statements they do not understand, e.g. comments, SQL `trigger` and `check`, etc). 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 7 (creation of a referential constraint between column C2 and table T1) is associated with this pattern, this procedure can be executed automatically (under user's control) for each instantiation of the pattern. In this way, the user can build a powerful custom tool which detects foreign keys in queries and which adds them in the schema automatically.

PDL and its engine are being extended to higher-level patterns, and to the expression of semantic constraints on the pattern variables. For instance, it will allow stating that, according to the repository contents, "*column A1 is a key of table A*", or "*the value of program variable X depends, directly or not, on the value of variable Y*".

DB-MAIN also includes a **name processor** with which selected names in a schema, or in selected objects of a schema, can be transformed according to substitution patterns. Some examples of patterns :

`"^C-" -> "CUST-"` *replaces all prefix "C-" by the prefix "CUST-";*

`"DATE" -> "TIME"` *replaces each substring "DATE", whatever its position, by the subtstring "TIME";*

`"^CODE$" -> "REFERENCE"` *replaces all the names "CODE" by the new name "REFERENCE".*

In addition, it allows 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 tool dedicated to solving a special kind of problems, or conducting specific activities. It gives access to the basic toolboxes of the tool in a controlled and intelligent way. DB-MAIN currently includes three assistants.

The transformation assistant

It 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 object is a many-to-many relationship type), and the solution is an action resulting in eliminating the problem (transform it into an entity type). Several dozens of problem/solution items are proposed. The user can select one of them, and execute it automatically or in a controlled way. Alternatively, he can build a script comprising a list of operations, execute it, save and load it. Predefined scripts are available to transform any schema according to popular models : Bachman model, binary model, relational, CODASYL, standard files, conceptualization of relational schemas (DBRE). Customized operations can be added via V2 functions (section 10).

The analysis assistant

This tool is dedicated to the analysis of schemas. The first step consists in defining a submodel as a restriction of the generic specification model described in section 5 (further detail can be found in [H3]). This restriction appears as a boolean expression of elementary predicates stating which specification patterns are valid. Some examples : "an entity type must have from 1 to 100 attributes", "a relationship type has from 2 to 2 roles", "the entity type names are less than 18-character long", "a name does not include spaces", "no names belong to a given list", "an entity type has from 0 to 1 supertype", "the schema is hierarchical", "there is 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 V2 functions (section 10).

The second step consists in evaluating the current schema against a specific submodel. This provides a list describing the violations detected.

The reverse enginering assistant

DBRE includes several complex and highly knowledge-based activities. This assistant includes separate modules dedicated to specific problems which appear in the processes discussed in section 2. Some examples :

- the DATA STRUCTURE EXTRACTION assistant proposes guidance functions to find identifiers, foreign keys, field refinement, value domains, according to various heuristics and information sources;
- the UNTRANSLATION assistant includes the rules related to the ER/DMS mappings according to the popular DMS : SQL, CODASYL, IMS, TOTAL/IMAGE, DATACOM-DB, standard files, etc.
- the DE-OPTIMISATION assistant knows about the main optimization heuristics, and can guide the analysts to detect and eliminate technical constructs.

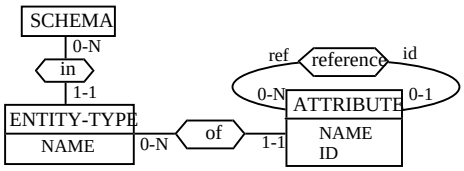
10. FUNCTIONAL EXTENSIBILITY

No CASE tool can satisfy the needs of all users in any possible situation. There are two important domains in which users require customized extensions, namely additional internal functions and interfacing with other tools. DB-MAIN provides a set of built-in standard tools. These

tools are sufficient to satisfy most basic needs in database engineering. However, specialized operators may be needed to deal with unforeseen or marginal situations. In addition, analysing and generating texts in any language and according to any dialect, or importing and exchanging specifications with any CASE tool or DDS is practically impossible, even with highly parametric import/export processors. To cope with such problems, DB-MAIN provides the VOYAGER-2 (V2) development environment allowing users to build their own functions, whatever their complexity. V2 offers a powerful language in which specific DB-MAIN tools can be developed. Basically, V2 is a procedural language allowing accessing and modifying the repository.

Its main features can be summarized as follows :

- communicating with the repository uses either predicative or navigational queries;
- functions and procedures can be recursive;
- generic, shared, list structures are provided, with powerful list operators; in particular, lists of repository objects can be built and processed; automatic garbage collection is provided;
- powerful input/output text functions allows easy development of parsing and generating functions;
- all the DB-MAIN basic tools are available from V2;
- a V2 procedure can be attached to DB-MAIN objects (dialog boxes, patterns, buttons, etc)
- a V2 procedure can appear in a DB-MAIN menu in the same way as basic tools (seamless functional extension);
- a V2 procedure is precompiled into an internal binary code. This code is interpreted by a virtual V2 machine.




```
function integer MakeForeignKey (string :
   T1,T2,C1,C2)
/* if C1 is an identifying attribute of entity type T1 and if C2 is
   an attribute of T2, and if C1 and C2 are compatible, then
   define C2 a foreign key to T1  */
   schema : S;
   entity_type : E;
   attribute : A,ID,FK;
   list : ALI,ALF;
{  S := GetCurrentSchema(); /* S = current schema
   */
/* ALI = list of the attributes (with name C1 and which are
   identifier) of the entity types in S with name T1  */
   ALI := attribute[A]
      {of:entity_type[E]{in:[S] and E.NAME =
   T1}
                          and A.NAME = C1
                          and A.ID = true};
/* ALF = list of the attributes (with name C2) of the entity types
   in S with name T2  */
   ALF := attribute[A]
      {of:entity_type[E]{in:[S] and E.NAME =
   T2}
                          and A.NAME = C2};
/* if both list are not-empty, then
    if the attributes are compatible then define the attribute in
   ALF as a foreign key to the attribute in ALI  */
   if not(empty(ALI) or empty(ALF))
   then {ID := GetFirst(ALI);
         FK := GetFirst(ALF);
         if ID.TYPE = FK.TYPE and
            ID.LENGTH = FK.LENGTH
         then {connect(reference,ID,FK);
               return true;}
         else {return false;};}
   else  {return false;};
}
```

Figure 7 - A (strongly simplified) excerpt of the repository and a VOYAGER-2 function which uses it. The input arguments are four names T1,T2,C1,C2 such as those resulting from an instantiation of the pattern of Figure 7. The function first evaluates the possibility for 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 figure 7 presents a small but powerful V2 function which validates and creates a referential constraint with the arguments extracted, e.g., from a COBOL/SQL program by the pattern of Figure 6.

11. METHODOLOGICAL CONTROL[7] AND DESIGN RECOVERY

Though this paper presents it as a DBRE CASE tool only, the DB-MAIN environment has a wider scope, i.e. data-based applications engineering. In particular, it is to address one of the most complex, but critical problem, namely application evolution. In this context, understanding how the engineering processes have been carried out when legacy systems have been developed, and guiding today 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**, still is in full development, and, so far, few results have been made available to practitioners. 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-

7 This part is the subject of the DB-Process subproject, and is fully supported by the *Communauté Française de Belgique*.

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 RE 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 developped. This design includes not only the specifications, but also the reasonings, the transformations, the hypotheses, the decisions which the development process was made of.

Briefly stated, DB-MAIN proposes a design process meta-model comprising concepts such as *design process, design strategy, design product, decision, hypothesis* and *justification*. A specific method (either forward or reverse) can be described as a set of processes and products. The DB-MAIN CASE tool is controlled by a **method engine** which is able to interprete such a method description. In this way, the tool is customized according to this specific method. 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 (as well as the reference method) is recorded in the repository. It can be examined, replayed, synthetized, and processed (e.g. for design recovery). A more comprehensive description of how these problems are addressed in the DB-MAIN approach and CASE tool can be found in [H6].

12. CONCLUSIONS

Considering the requirements sketched in section 3, few (if any) commercial CASE 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 analyzing the data structure declaration statements. No help is provided in 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 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 applying non standard translation rules, and including performance oriented constructs. Current DBRE CASE 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. In other words, the current CASE tools enforce a DBRE methodology which can be described as a specialization of the generic model of section 2 in which the DS Extraction process is reduced to DMS-DDL text ANALYSIS, and the DS Conceptualization process is reduced to SCHEMA UNTRANSLATION and (possibly) SCHEMA NORMALIZATION..

The DB-MAIN CASE tool presented in this paper tries to go a step further by addressing some of the problems and information sources currently ignored. Though it still is in development (the final version is due in 1997), it already is successfully used in several companies to process actual complex applications. These applications return essential feedback on the usability of each functions, on their degree of usefulness, and on the completeness and soundness of the approach developed in the DB-MAIN project.

In its current version, DB-MAIN includes the following features (as by January, 1995):

- multischema projects, conceptual, logical and physical database schemas,
- six schema presentation formats
- deriving logical schemas from conceptual schemas,
- deriving conceptual schemas from logical schemas,
- 20 transformations (for normalization, optimization, restructuring, implementation, reverse engineering, etc),
- transformation and analysis assistants (with scripting feature)
- generating executable code according to several DBMS,
- generating reports,
- automatic extractors for SQL, COBOL and CODASYL DDL source texts (IMS and DATACOM-DB due soon)
- PDL pattern matching engine (kernel), updating a logical schema from a source text, schema/source text hot links, program variables dependency graph, name processor,
- VOYAGER-2 environment (kernel),
- logging user's actions, and replaying them selectively,
- import/export.

Some additional components will be soon available, such as additional schema transformations and the DBRE assistant.

DB-MAIN has been developed in C++ for MS-Windows machines. In order to develop contacts and collaboration, an educational version (complete but limited to small applications) and its documentation are available for Windows 3.1/3.11. This free version can be obtained by contacting `jlhainaut@info.fundp.ac.be`.

13. REFERENCES

[A1] Andersson, M., *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering*, in

Proc. of the 13th Int. Conf. on ERA, Manchester, Springer-Verlag, 1994

[B1] Batini, C., Ceri, S., Navathe, S., B., *Conceptual Database Design*, Benjamin/ Cummings, 1992

[B2] Batini, C., Di Battista, G., Santucci, G., *Structuring Primitives for a Dictionary of Entity Relationship Data Schemas*, IEEE TSE, 19(4), 1993

[B3] Bolois, G., Robillard, P., *Transformations in Reengi-neering Techniques*, in Proc. of the 4th Reengineering Forum "Reengineering in Practice", Victoria, Canada, 1994

[C3] Chiang, R., H., Barron, T., M., Storey, V., C., *Reverse Engineering of Relational Databases : Extraction of an EER model from a relational database*, Journ. of Data and Knowledge Engineering, 12(2), pp107-142, 1994

[D3] Davis, K., H., Arora, A., K., *Converting a Relational Database model to an Entity Relationship Model*, in Proc. of ERA : a Bridge to the User, North-Holland, 1988

[F1] Fikas, S., F., *Automating the transformational development of software*, IEEE TSE, Vol. SE-11, pp1268-1277, 1985

[F3] Fonkam, M., M., Gray, W., A., *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, 1992

[H1], Hainaut, J.-L., *Theoretical and practical tools for data base design*, in Proc. Int. VLDB conf., ACM/IEEE, 1981

[H2] Hainaut, J-L., *Entity-generating Schema Transformation for Entity-Relationship Models*, in Proc. of the 10th ERA, San Mateo (CA), North-Holland, 1991

[H3] Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O., *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, 1992

[H4] Hainaut, J-L., Chandelon M., Tonneau C., Joris M., *Contribution to a Theory of Database Reverse Engineering*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, May 1993, IEEE CSP, 1993

[H5] Hainaut, J-L, Chandelon M., Tonneau C., Joris M., *Transformational techniques for database reverse engineering*, in Proc. of the 12th Int. Conf. on ERA, Arlington-Dallas, Springer-Verlag, LNCS, 1993

[H6] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D., *Evolution of database Applications : the DB-MAIN Approach*, in Proc. of the 13th Int. Conf. on ERA, Manchester, Springer-Verlag, 1994

[H7] *Software Reuse and Reverse Engineering in Practice*, Hall, P., A., V. (Ed.), Chapman&Hall, 1992

[I1]*Special issue on Reverse Engineering*, IEEE Software, January, 1990

[J1] Johannesson, P., Kalman, K., *A Method for Translating Relational Schemas into Conceptual Schemas*, in Proc. of the 8th ERA, Toronto, North-Holland, 1990

[K1] Kobayashi, I., *Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence*, in Information Systems, Vol. 11, No 1, pp. 41-59, January, 1986

[K2] Kozaczynsky, Lilien, *An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation*, in Proc. of ERA, 1987

[M1] Markowitz, K., M., Makowsky, J., A., *Identifying Extended Entity-Relationship Object Structures in Relational Schemas*, IEEE Trans. on Software Engineering, Vol. 16, No. 8, 1990

[N2] Navathe, S., B., Awong, A., *Abstracting Relational and Hierarchical Data with a Semantic Data Model*, in Proc. of ERA : a Bridge to the User, North-Holland, 1988

[N3] Nilsson,E., G., *The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema*, in Proc. of ERA, October, IEEE/North-Holland, 1985

[P1] Petit, J-M., Kouloumdjian, J., Bouliaut, J-F., Toumani, F., *Using Queries to Improve Database Reverse Engineering*, in Proc. of the 13th Int. Conf. on ERA, Manchester, Springer-Verlag, 1994

[P2] W.J. Premerlani, W., J., Blaha, M.R., *An Approach for Reverse Engineering of Relational Databases*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, IEEE CSP, May, 1993

[R2] Rosenthal, A., Reiner, D., *Theoretically sound transformations for practical DB design*, in Proc. of ERA, 1988

[R3] Rosenthal, A., Reiner, D., *Tools and Transformations - Rigourous and Otherwise - for Practical Database Design*, ACM TODS, Vol. 19, No. 2, June 1994

[S3] Shoval, P., Shreiber, N., *Database Reverse Engineering : from Relational to the Binary Relationship Model*, Data and Knowledge Engineering, Vol. 10, No. 10, 1993

[S4] Signore, O, Loffredo, M., Gregori, M., Cima, M., *Reconstruction of ER Schema from Database Applications: a Cognitive Approach*, in Proc. of the 13th Int. Conf. on ERA, Manchester, Springer-Verlag, 1994

[S5] Springsteel, F., N., Kou, C., *Reverse Data Engineering of E-R designed Relational schemas*, in Proc. of Databases, Parallel Architectures and their Applications, March, 1990

[T1] Teorey, T. J., *Database Modeling and Design : the Fundamental Principles*, Morgan Kaufmann, 1994

[W1] Weiser, M., *Program Slicing*, IEEE TSE, Vol. 10, 1984, pp 352-357