Database Reverse Engineering

Jean-Luc Hainaut
University of Namur, Belgium

Jean Henrard
ReveR s.a., Belgium

Didier Roland
ReveR s.a., Belgium

Jean-Marc Hick
ReveR s.a., Belgium

Vincent Englebert
University of Namur, Belgium

1. INTRODUCTION

Database reverse engineering is that part of Information System Engineering that addresses the problems and techniques related to recovering abstract descriptions of files and databases of legacy systems. According to (Brodie, 1995), a Legacy Information System can be defined as [a] *data-intensive application, such as [a] business system based on hundreds or thousands of data files (or tables), that significantly resists modifications and changes*. The objective of data reverse engineering can then be sketched as follows: to recover the logical and conceptual descriptions, or schemas, of the permanent data of a legacy information system, i.e., its database, be it implemented as a set of files or through an actual database management system.

By *logical schema*, we mean the description of the technology-dependent database structures that would be required by a programmer in order to develop a new application on the legacy data. The *conceptual schema* of the data is an abstract, technology-independent description of what these data structures mean, i.e., their semantics.

Basically, database reverse engineering seldom is a goal *in se*, but most often is one of the first steps in a broader engineering project. Indeed, rebuilding the precise documentation of a legacy database is an absolute prerequisite to any attempt to migrate, reengineer, maintain or extend it, or to merge it with other databases.

When one analyzes the commercial offering in CASE support for database reverse engineering, one gets the feeling that the problem has been strongly exaggerated. Generally, it reduces to the derivation of a conceptual schema such as that of Figure 1 from the following DDL code, which seems to be quite a straightforward process.

```
create table CUSTOMER (
   CNUM decimal(10) not null,
   CNAME varchar(60) not null,
   CADDRESS varchar(100) not null,
 primary key (CNUM))
```

```
create table ORDER (
   ONUM decimal(12) not null,
   SENDER decimal(10) not null,
   ODATE date not null,
 primary key (ONUM),
 foreign key (CNUM) references CUSTOMER))
```

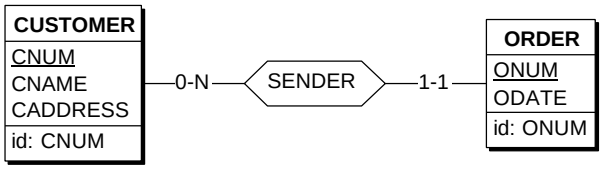

Fig. 1. A naive view of data reverse engineering

Unfortunately, *actual* database reverse engineering often is closer to deriving the conceptual schema of Figure 2 from the following sections of COBOL code, using meaningless names and that do not declare compound fields nor foreign keys.

```
select CF008 assign to DSK02:P12
 organization is indexed
 record key is K1 of REC-CF008-1.
select PF0S assign to DSK02:P27
 organization is indexed
 record key is K1 of REC-PFOS-1.
fd CF008.
 record is REC-CF008-1.
 01 REC-CF008-1.
    02 K1 pic 9(6).
    02 filler pic X(125).
```

```
fd PF0S.
 records are REC-PF0S-1,REC-PF0S-2.
 01 REC-PF0S-1.
    02 K1.
       03 K11 pic X(9).
       03 filler pic 9(6)
    02 filler pic X(180).
 01 REC-PF0S-2.
    02 filler pic X(35).
```

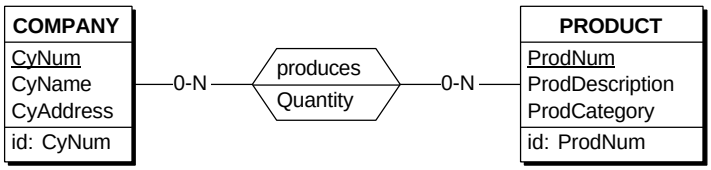

Fig. 2. A more realistic view of data reverse engineering.

Getting such a result obviously requires additional sources of information, which may prove more difficult and tedious to analyze than mere DDL statements. Untranslated (implicit) data structures and constraints, non standard implementation approaches and techniques, optimization constructs, ill-designed schemas and, above all, the lack of up to date documentation are some of the difficulties that the analysts will face when trying to understand existing databases from operational system components.

The goal of this paper is to describe the problems that arise when one tries to rebuilt the documentation of a legacy database and the methods, techniques and tools through which these problems can be solved. A more in-depth analysis of this domain can be found in (Hainaut, 2002).

2. BACKGROUND: STATE OF THE ART AND KEY PROBLEMS

Database reverse engineering has been recognized to be a specific engineering problem in the eighties, notably in (Casanova, 1984), (Davis, 1985) and (Navathe, 1988). These pioneering approaches were based on simple and intuitive rules such as those illustrated in Figure 1. They worked quite nicely with databases designed in a clean and disciplined way. A second generation of methodologies coped with physical schemas resulting from empirical design that could no longer be qualified simple and intuitive. More complex design rules were identified and interpreted (Blaha, 1995), structured and comprehensive approaches were developed (Hainaut, 1993) (Edwards, 1995) and the first industrial tools appeared (e.g., Bachman's Reengineering Tool). Many contributions were published in the nineties, addressing practically all the legacy technologies and exploiting in a sophisticated way several sources of information such as application source code, the database contents or the

application user interface. Among synthesis publications, let us mention (Davis, 2000), the first tentative history of this discipline.

These second generation approaches were faced with two families of problems that we will briefly discuss here below, namely the elicitation of implicit constructs and the semantic interpretation of logical schemas.

The *Implicit construct* problem

One of the hardest problems *reverse engineers* have to face is eliciting implicit constructs. By this expression, we mean data structures or data properties, such as integrity constraints, that are an integral part of the database, though they have not been explicitly declared in the DDL specifications, either because they have been discarded during the implementation phase, or because they have been implemented by other means, such as through validation code in the application programs. Let us shortly describe two common patterns.

Implicit field/record structure. Ideally, all the record fields of the database should be identified and given meaningful names. However, compound fields or even whole record structures often are expressed as anonymous or unstructured fields, so that their intended structure is left implicit. The following DDL code sample shows, at the left side, the code that declares the database record type CUSTOMER, the actual intended structure of which is at the right side. The application programs generally recover the actual structure by storing records in local variables that have been given the correct detailed decomposition.

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

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

Fig. 3. At the left, the actual DDL code describing a record type structure; at the right, the intended (implicit) code.

Implicit foreign keys. Inter-record associations most often are translated into foreign keys in relational databases or into navigable parent/child associations in hierarchical and network databases. Databases made up of standard files also include foreign keys, that is, fields used to identify a record in another file, but these keys cannot be declared in the file system DDL and are left *implicit*. It may come as a surprise that hierarchical, network and even recent relational databases often include many undeclared, implicit foreign keys. There are several reasons to that, such as awkward or lazy design, but also backward compatibility (e.g, compliance with Oracle 5, which ignores primary and foreign keys), and non-standard ways to control referential integrity (e.g., simulation of a delete mode that is not available). Generally, the application programs include code fragments that ensure that data states that violate referential integrity are identified and coped with. For instance each statement that stores an ORDER record is preceded by a short code section that verifies that the CUSTOMER file includes a record for the customer that issued the order.

Semantic interpretation of logical constructs

Once the logical schema has been rebuilt, including all the implicit constructs, the problem arises to recover its intended semantics, or, in database words, its conceptual schema. The process, called interpretation or conceptualization, is fairly straighforward for small and simple database schemas, but it can prove very complex for large and ageing databases.

This process appears to be the inverse of the logical design of a database, the process through which a conceptual schema is translated into a technology-dependent model, such as a relational schema. Since logical design has been widely described in the literature (Batini, 1992) as a sequence of

transformation rules, it seems that the conceptualization process could be defined by reversing these transformations (Hainaut, 2006). This is valid to a large extent, provided we know which rules have been applied when the database was developed. What makes this idea more complex than expected is that many legacy databases have been developed through empirical approaches, based on unusual and sometimes *esoteric* rules (Blaha, 1995). We illustrate the conceptualization process through two common but non trivial examples of reverse transformations.

Recovering ISA relations. Since most legacy data management systems do not include explicit representations of supertype/subtype hierarchies, the latter must be converted into equivalent but simpler constructs. One of the most popular translation techniques consists in implementing the bottom-most entity types through downward inheritance. Rebuilding the hierarchies from this implementation can be performed by such techniques as *formal concept analysis* (Godin, 2000), as shown in Figure 4.

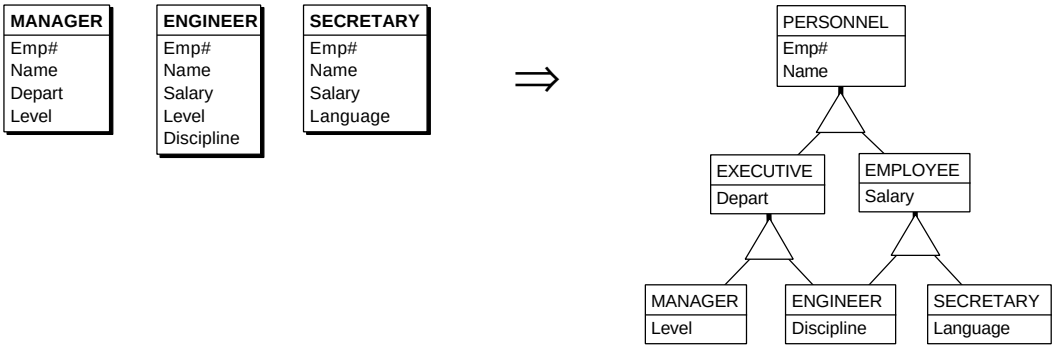

Fig. 4. Recovering an ISA hierarchy from record types sharing common fields

Recovering relationship types. There are many different techniques to implement relationship types in logical models. This is particularly true for complex relationship types, that is, those which are n-ary, many-to-many or those that have attributes and that are submitted to additional integrity constraints. Figure 5 shows a typical implementation of a many-to-many relationship type with attributes as a multivalued, compound field incorporated in one of its members. The source relationship type is recovered through a complex transformation. Though this implementation pattern is quite common in COBOL record types, it has also been observed in relational database, where the complex field is implicit, and must first be recovered as described in Figure 3.

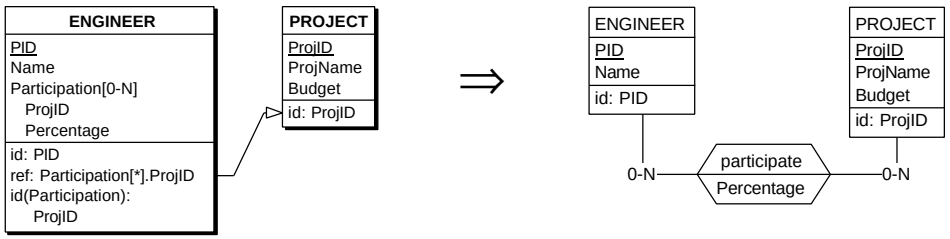

Fig. 5. Expressing a complex multivalued compound attribute as a many-to-many relationship type

3. THE PROCESSES OF DATABASE REVERSE ENGINEERING

Grossly speaking, the reverse engineering of a database appears to consist in reversing its design process (Baxter, 2000). The latter can be decomposed into three main phases, namely *logical design*, which transforms the conceptual schema into a DBMS-dependent logical schema, *physical design*,

through which technical constructs, such as indexes, are added and parametrized, and *coding*, which translates the physical schema into DDL code. Starting from the DDL code, as well as from other information sources, rebuilding the conceptual schema can also be performed in three phases: physical schema extraction (the inverse of the coding process), logical schema extraction (the inverse of physical design) and conceptualization (the inverse of logical design).

In empirical approaches, the standard design phases are not so clearly identified and often are interleaved. This makes reverse engineering both more complex (each construct can implement several concerns) and less deterministic (the same physical schema can lead to several conceptual schemas).

In this section, we first discuss the architecture of a reverse engineering project. Then, we describe the three main processes identified here above. Finally, we discuss the role of reverse engineering tools.

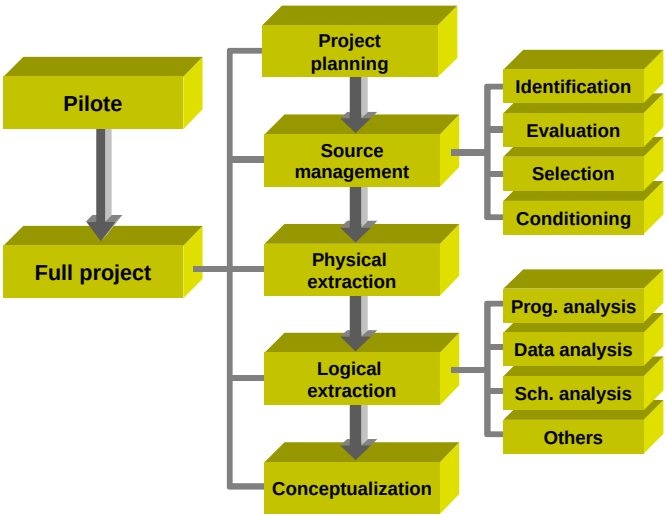

Fig. 6. A typical data reverse engineering project

Database reverse engineering projects

Large-scale database reverse engineering projects typically are risky processes, so that carrying out a pilote project first is common practice. Indeed, it provides precise measures on the quality of the information sources, on the quality of the database, on the motivation of the staff in charge of the database, on the skills required and finally on the resources needed, in terms of money, time and manpower (Aiken, 1996).

The project itself includes two preliminary phases, namely project planning and management of the information sources (Figure 6). The three next phases are those mentioned here above and are of technical nature.

Physical structure extraction

Physical extraction is aimed at building the physical schema that is explicitly declared by the DDL code (or the data dictionary contents) of the database (Fig. 7). ~~Generally,~~ This process brings no additional value to the schema, but transforms it in a form that allows further processing. This extraction generally uses a parser that stores the physical data structures in some repository. For instance, the contents of the DDL code is expressed in a graphical view that shows (for a relational database) the tables, the columns, the primary, secondary and foreign keys, the indexes, the table spaces, the check predicates, the triggers and the stored procedures.

This process can be more complex when useful information has also been coded in views instead of in the global schema. This will be the case for standard files (for which there is no global schema but a

large collection of record descriptions included in the program source code), for CODASYL databases (with their *subschemas*) and for relational databases (with their relational *views*). This multiplicity of partial schemas requires an integration phase that applies to the extracted schemas.

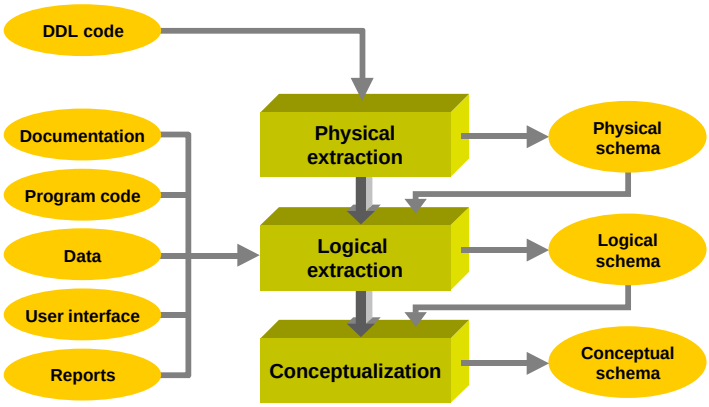

Fig. 7. The information sources, the main processes and the products of data reverse engineering

Logical structure extraction

The goal of this process is to recover the complete logical schema of the database, including all the implicit and explicit structures and constraints. The starting point is the physical schema that has been extracted from the DDL code. It is then enriched with the implicit constructs that are recovered through various analysis techniques. Actual case studies have shown that there often are more implicit constructs than explicit ones. This means that the DDL code, and therefore the physical schema as well, conveys less than half the semantics of the logical schema. Hence the importance of this phase, which too often is overlooked both in the literature and in commercial CASE proposals.

The most important constructs that have to be made explicit are the exact *field and record structures* (as illustrated in Fig. 3), the *unique keys* of record types and of multivalued fields and the *foreign keys*. Fig. 5 provides an illustration of the latter two concepts. The subfield ProjID of compound field Participation in record type ENGINEER plays two important roles. First, its values are unique among all the Participation instances of a given ENGINEER record. Secondly, each of its values references a PROJECT record. Therefore, this field acts as an identifier for the values of multivalued attribute Participation and as a multivalued foreign key targeting PROJECT. Needless to say that no DDL can express these properties, so that they basically are implicit. Additional constructs can be looked for as well, such as functional dependencies, value domains, optional fields or more meaningful record and field names.

Several information sources and techniques can be used to recover implicit constructs. Let us mention some of them.

- *The physical schema*. Missing data structures and constraints can be inferred from existing structural patterns. For instance, names can suggest roles (unique keys, foreign key), data types or relationships between data.
- *Technical/physical constructs.* There can be some correlation between logical constructs and their technical implementation. For instance, a foreign key is often supported by an index. Therefore, an index can be ~~an~~ evidence that a field could be a foreign key.
- *Application programs.* The way data are used, transformed and managed in the application programs brings essential information on the structural properties of these data. Analyzing the source code of these programs requires specific techniques and tools. Dataflow analysis, dependency analysis, programming *cliché* analysis and program slicing (Clève, 2006) are some examples of useful program processing techniques borrowed from the *program understanding* discipline (Henrard, 2003). The fine-grained analysis of DML statements can provide hints on data structures properties. See for instance (Andersson, 1994) for COBOL statements analysis

and (Petit, 1994) for SQL statement analysis.

- *Screen/form/report layout*. A screen form or a structured report can be considered derived views of the data. The layout of the output data as well as the labels and comments can bring essential information on the data (Heeseok, 2000).
- *External data dictionaries and CASE repositories*. Third-party or in-house data dictionary systems allow data administrators to record and maintain essential descriptions of the information resources of an organization, including the file and database structures. The same can be said of CASE tools, that can record the description of database structures at different abstraction levels.
- *Data*. The data themselves can exhibit regular patterns, uniqueness or inclusion properties, that provide hints that can be used to confirm or disprove structural hypotheses. The analyst can find evidence that suggests the presence of identifiers, foreign keys, field decomposition, optional fields or functional dependencies (Novellia, 2001). Good approximation of enumerated value domains can often be computed.
- *Non-database sources*. Small volumes of data can be implemented with general purpose softwares such as spreadsheet and word processors. In addition, semi-structured documents are increasingly considered a source of complex data that also need to be reverse engineered. Indeed, large text databases can be implemented according to representation standard such as SGML, XML or HTML that can be considered special purpose DDL.
- And of course, the *current documentation*, if any.

Data structure conceptualization

This process extracts from the complete logical schema a tentative conceptual schema that represents the likeliest semantics of the former (Fig. 7). It mainly relies on transformational techniques that *undo* the effect of the translation rules through which the logical schema was (or should have been) derived from the conceptual schema when the database was designed.

This complex process is decomposed in three subprocesses, namely untranslation, de-optimization and conceptual normalization.

The *untranslation* process consists in reversing the transformations that (are supposed to) have been used to draw the logical schema from the conceptual schema. For instance, each foreign key is interpreted as the implementation of a many-to-one relationship type. This process relies on a solid knowledge about the rules and heuristics that have been used to design the database. These rules can be quite standard, which makes the process fairly straighforward, but they can also be specific to the company or even to the developer in charge of the database, in which case, reverse engineering can be quite tricky, specially if the developer has left the company. The main constructs that have to be recovered are relationship types (see Fig. 5), super-type/subtype hierarchies (see Fig. 4), multivalued attributes, compound attributes and optional attributes.

The *de-optimization* process consists in removing from the schema the trace of all the optimization techniques that have been used to improve the performance of the database. Among the most common optimization patterns, let us mention the redundancies, that have to be identified and discarded, the unnormalized data structures, that must be split, and horizontal and vertical partitioning that must be identified and undone.

The goal of the *conceptual normalization* process is to improve, if necessary, the expressiveness, the simplicity, the readability and the extendability of the conceptual schema. Being quite common in conceptual analysis (see (Batini, 1993) for instance), we do not describe it in more detail.

Database reverse engineering tools

Due to the complexity and to the size of the information sources, database reverse engineering cannot be performed without the support of specific tools. An analysis of the requirements has been detailed in (Hainaut, 1996). So far, there is no commercial specific data-oriented Computer-Aided Reverse Engineering (CARE) tools. This is not a drawback anyway, since the reverse engineering process

shares much, in terms of models and techniques, with standard engineering activities. Unfortunately only limited reverse engineering functions can be found in current CASE tools such as Power-Designer, AMC-Designor, Rational Rose or Designer 2000. At best, they include elementary parsers for SQL databases, foreign key elicitation under very strong assumptions (primary and foreign keys must have the same names and types) and some primitive standard foreign key transformations. None can cope with complex reverse engineering projects. The most advanced tool probably is DB-MAIN, an experimental CASE tool (http://www.info.fundp.ac.be/libd) that includes components that address some of the complex issues discussed in this article, such as code parsing, code analysis, schema analysis and schema transformation.

4. TRENDS AND PERSPECTIVES

Database reverse engineering most often is the first step of larger-scope projects such as system reengineering or system migration. It appears that the information collected during the reverse engineering process can help carrying out the next steps. We illustrate this idea with *system migration*, through which a legacy database as well as all its application programs are ported to a modern platform. The formal mapping between the (legacy) source schema and the (new) target schema includes enough information to automatically generate the data migration procedures, so that the new database can be fully converted in the new technology (Hick, 2001). In addition, this mapping also provides enough information to automatically restructure the legacy application programs so that they now run against the new database (Clève, 2006b).

The second trends that has been shaping important development in reverse engineering concerns techniques that address other types of data structures such as semi-structured data. In particular, web reverse engineering techniques have been proposed for several years to extract structured data from web pages, generally through web wrappers that are built by reverse engineering techniques (Chung, 2002).

5. CONCLUSIONS

Many reverse engineering problems are now identified and formally described. For most of them, we are provided with techniques to solve them, at least partially. However, solving them in every possible situation is a goal that is far from being reached. In addition, some problems still remain to be worked out. We mention three of them.

- The economics of database reverse engineering is a critical point for which there is no clear proposal so far. In particular, we do not know at the present time how to evaluate the cost of a reverse engineering project and when to stop such a project, either because we have got enough information or because it proves too costly.
- The scalability of most techniques that have been proposed so far has yet to be demonstrated. Searching a 30,000 LOC program for a specific programming *cliché* is quite realistic. The same cannot be asserted for a 3,000,000 LOC set of programs. In the same way, finding PK/FK patterns in a physical schema based on name and data type similarity is reasonably fast in a 200-table schema, but may be desperately slow in a tenfold larger schema.
- Though reverse engineering legacy databases 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 than it was in the past. However, despite this maturity, much remains to be done to transfer its results among practitioners.

6. REFERENCES

Aiken, P. (1996). *Data Reverse Engineering, Slaying the Legacy Dragon*, McGraw-Hill.

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*, Springer-Verlag

Batini, C., Ceri, S., Navathe, S. (1992). *Conceptual Database Design - An Entity-Relationship Approach*, Benjamin/ Cummings

Baxter, I., Mehlich, M. (2000). Reverse engineering is reverse forward engineering, Science of Computer Programming, 36 (2000) 131-147, Elsevier

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

Brodie, M., Stonebraker, M. (1995). *Migrating Legacy Systems*, Morgan Kaufmann.

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.*, 28(1)

Chung, C., Gertz, M., Sundaresan, N. (2002). Reverse Engineering for Web Data: From Visual to Semantic Structures, Proc. of the 18th Int. Conf. on Data Engineering (ICDE'02), IEEE Computer Society Press.

Clève, A., Henrard, J., Hainaut, J.-L. (2006). Data Reverse Engineering using System Dependency Graphs, in *Proc. of WCRE'06*, , IEEE Computer Society, 2006

Clève, A. (2006b) Co-transformations in Database Applications Evolution, in *Generative and Transformational Techniques in Software Engineering*. Ralf Lämmel, João Saraiva, Joost Visser, eds, LNCS 4143, Springer-Verlag, 399-411, 2006

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., Aiken, P., H. (2000). Data Reverse Engineering: A historical View, in *Proc. of the 7th Working Conference on Reverse Engineering (WCRE'00)*, IEEE Comp. Soc. Press., 2000

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

Godin, R, Mili H., Mineau G., Missaoui R., Arfi A., Chau, T. (2000). Design of class hierarchies based on concept, (Galois) lattices. In *Theory and Practice of Object Systems*, 4(2) , 117-134, Wiley

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

Hainaut, J.-L. (2002). *Introduction to database reverse engineering* In LIBD lecture notes, Pub. University of Namur, Belgium, 160 pages. Retrieved August 2005 from http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf

Hainaut, J-L. (2006). The transformational approach to database engineering, in *Generative and Transformational Techniques in Software Engineering*. Ralf Lämmel, João Saraiva, Joost Visser, eds, LNCS 4143, Springer-Verlag, 89-138, 2006

Hainaut, J-L, Roland, D., Hick J-M., Henrard, J., Englebert, V. (1996). Database Reverse Engineering: from Requirements to CARE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 1 (1996).

Heeseok, L., Cheonsoo, Y. (2000). A Form Driven Object-oriented Reverse Engineering Methodology. in *Information Systems* 25(3), 235-259.

Henrard, J. (2003). Program understanding in database reverse engineering, PhD Thesis, University of Namur, 2003 http://www.info.fundp.ac.be/~dbm/publication/2003/jhe_thesis.pdf

Hick, J.-M. (2001). Evolution of relational Database Applications (in French), PhD Thesis, University of Namur, 2001 http://www.info.fundp.ac.be/~dbm/publication/2001/these-jmh.pdf

Novellia, N., Rosine Cicchetti R. (2001). Functional and embedded dependency inference: a data mining point of view, Information Systems, 26 (2001), 477–506

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

7. KEY TERMS

Database Reverse Engineering: The process through which the logical and conceptual schemas of a legacy database, or of a set of files, are recovered, or rebuilt, from various information sources such as DDL code, data dictionary contents, database contents, or the source code of application programs that use the database.

Implicit construct: A data structure or an integrity constraint that holds, or should hold, among the data, but that has not been explicitely declared in the DDL code of the database. Implicit compound and multivalued fields as well as implicit foreign keys are some of the most challenging constructs to chase when recovering the logical schema of a database.

Data Structure Extraction: The process within Reverse Engineering that attempts to recover all, or at least most, implicit data structures and integrity constraints.

Data Structure Conceptualization: The process within Database Reverse Engineering that aims at deriving a plausible conceptual schema from the logical schema of a legacy database. Also called *Schema Interpretation*.

Optimization schema construct. Any data structure or constraint that appears in a logical or physical schema and that conveys no information. Its aim is to increase the performance of the database and of its application programs that use it. An optimization construct has no counterpart in the conceptual schema. Unnormalized tables, redundancies, artificially split or merged tables are some popular examples.

Legacy information system. *A data-intensive application, such as [a] business system based on hundreds or thousands of data files (or tables), that significantly resists modifications and change (*Brodie, 1995*).*

System empirical design. A way to build a system that does not rely on a strict methodology but rather on the experience and the intuition of the developer. An empirically designed database often includes undocumented idiosyncrasies that may be difficult to understand when attempting to reverse engineer it.