Program Understanding in Databases Reverse Engineering

J. Henrard, V. Englebert, J-M. Hick, D. Roland, J-L. Hainaut

Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur
db-main@info.fundp.ac.be

Abstract. The main argument of the paper is that *database understanding* (or reverse engineering) requires sophisticated *program understanding* techniques, and conversely. Database reverse engineering (DBRE) can be carried out following a generic methodology, one of the phases of which consists in eliciting all the implicit and untranslated data structures and constraints. Evidences of these hidden constructs can be found by analysing how the programs use and update the data. Hence the need for program analysis techniques such as searching for *clichés*, dependency analysis, program slicing and synthetic views. The paper explains how these techniques contribute to DBRE, and describes DB-MAIN, a programmable and extensible CASE environment that supports DBRE through program understanding techniques.

1. Introduction

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. Recovering these specifications is generally intended to redocument, convert, restructure, maintain or extend legacy 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. Therefore, reverse engineering has long been recognised 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 the central component of which is a database (or a set of permanent files), it is generally considered that the complexity can be broken down by considering that the files or databases can be reverse engineered (almost) independently of the procedural parts, through a process called *DataBase Reverse Engineering* (DBRE in short).

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 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 application data components first can be much more efficient than trying to cope with the whole application.

Even if reverse engineering the data structure is "easier" than recovering the specification of the application as a whole, it is still a complex and long task. Many techniques, proposed in the literature and offered by CASE tools, appear to be limited in scope and are generally based on unrealistic assumptions about the quality and completeness of the source data structures to be reverse engineered. For instance, they often suppose that all the conceptual specifications have been declared into the DDL (data description language) and the procedural code is ignored. The schema has not been deeply restructured for performance or for any other requirements. Names have been chosen rationally.

Those conditions cannot be assumed for most large operational databases. Since 1992, some authors have recognised that the procedural part of the application programs is an essential source of information to retrieve data structures ([1], [4], [7] and [8]), and that understanding some program aspects is one of the keys to fully understand the data structures.

Program understanding is an emerging area within the software engineering field. It is the process of acquiring knowledge about an existing, generally undocumented, computer program. Increased knowledge enables such activities as error correction, enhancement, reuse, documentation and reverse engineering. A variety of approaches have been investigated to analyse programs, from straightforward textual analysis, through increasingly more complex static approaches, to the dynamic analysis of executing programs [9].

In this paper, we explain why program understanding can be useful to help understand data structures as well, and how these techniques can be used in DBRE. The paper is organised as follows. Section 2 is a synthesis of a generic DBMS-independent DBRE methodology. Section 3 discusses the need of program understanding in DBRE and the information we are looking for in programs. Section 4 describes the techniques used to analyse programs. Section 5 presents a DBRE CASE tool which is intended to support most DBRE processes, including program understanding.

2. A generic methodology for database reverse engineering

Our database reverse engineering (DBRE) methodology [4] is divided into two major processes (Fig. 1), namely *data structure extraction* and *data structure*

conceptualisation. Those problems correspond to recovering two different schemas and require different concepts, reasoning and tools. In addition, these processes grossly appear as the reverse of the physical and logical design usually admitted in database design methodologies [2].

This methodology is generic in two ways. 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.

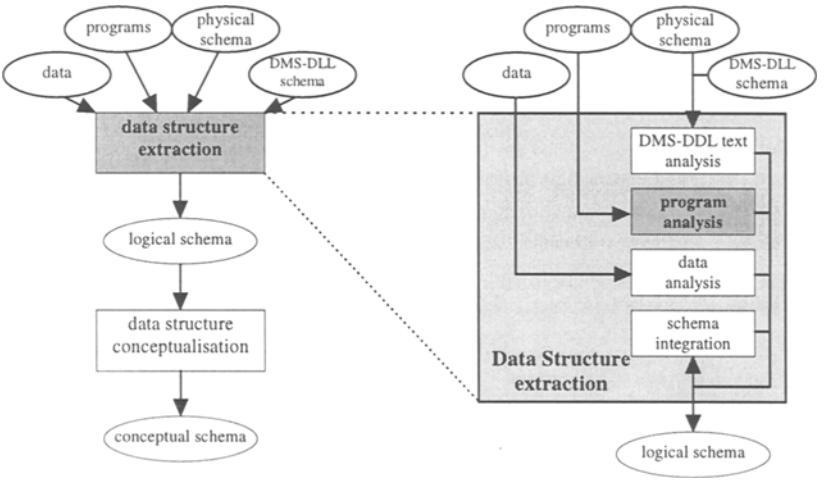


Fig. 1. Main processes of the generic DBRE methodology.

2.1 Data structure extraction

The first phase of the methodology consists in recovering the complete DMS schema, including all the implicit and explicit structures and constraints.

Some DMS, mainly the DBMS, supply a description of the global data schema. The problem is much more complex for standard files, for which no computerised description of their structure exists in most cases. The analysis of the declaration statements of each source program provides a partial view of the file and record structures. Data records and fields declaration statements provide important but partial information only. Hence the need for program and data analysis and for integration techniques.

If a construct or a constraint has not been declared explicitly in the database schema, whatever the reason, it has most of the time been translated into procedural code sections distributed and duplicated throughout the programs. Recovering undeclared, and therefore implicit, structure is a complex problem, for which no deterministic methods exist so far. A careful analysis of all the information sources (procedural sections, documentation, database content, etc.) can accumulate evidences for those specifications.

In this generic methodology, the main processes of data structure extraction are the following :

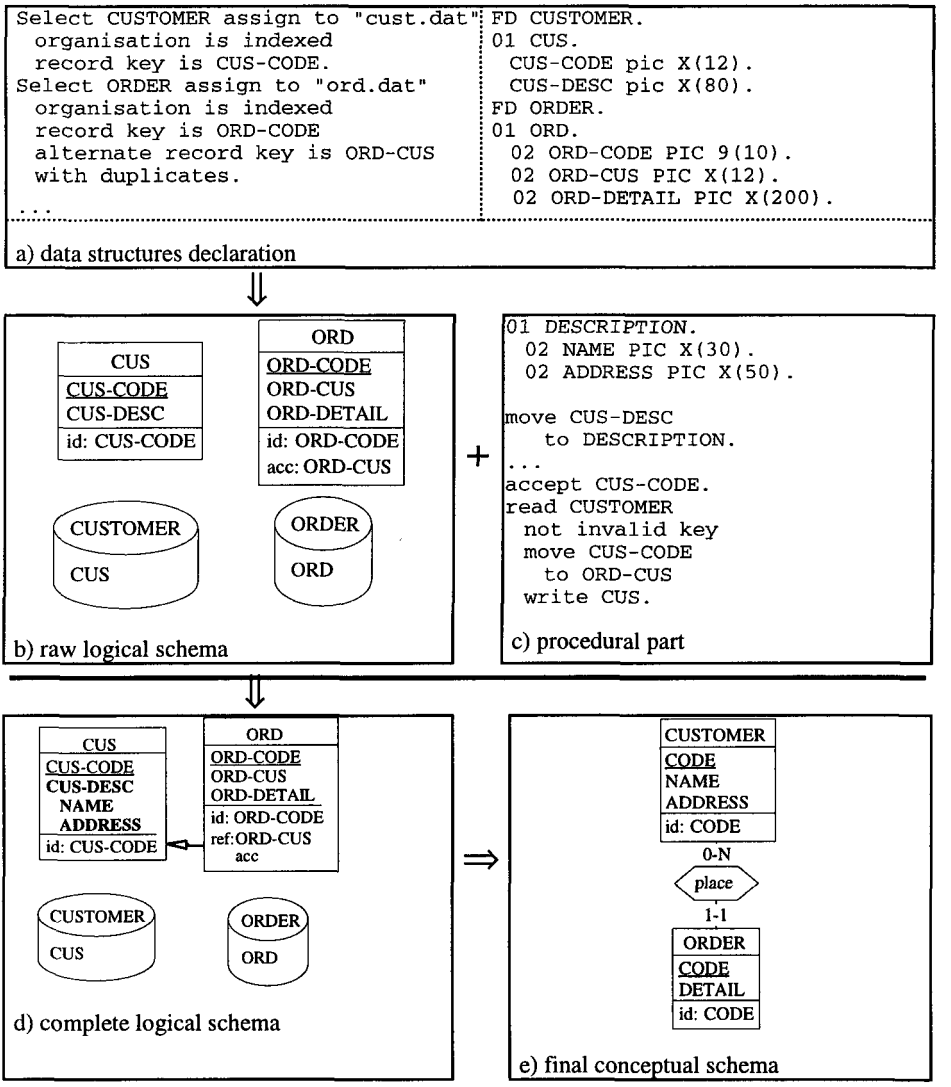


Fig. 2. Database reverse engineering example.

- *DMS-DDL text analysis*: analysing the data structure declaration statements to extract the explicit constructs and constraints, thus providing a rows logical schema.
- *Program analysis*: analysing the procedural code to detect implicit data structures.
- *Data analysis*: analysing the data in order to detect data patterns suggesting data structures and properties and to test hypotheses.
- *Schema integration*: merging the schemas obtained during the previous steps.

The final product of this phase is the complete logical schema, that includes both explicit and implicit structures and constraints.

2.2 Data structure conceptualization

The second phase addresses the conceptual interpretation of the logical schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimisations and DMS-dependent constructs. The final product of this phase is the persistent data conceptual schema of the application. More detail can be found in [5].

Fig. 2 gives a short example of a DBRE process. The files and records declarations (2.a) are analyzed to yield the raw logical schema (2.b). This schema is refined through the analysis of the procedural parts of the program (2.c) to produce a complete logical schema (2.d). This schema exhibits two new constructs. It is then transformed into the conceptual schema (2.e).

3. Why program understanding in DBRE

The data structure extraction process is often easier for true databases than for standard files. Indeed, databases have a global schema (generally a DDL text or Data dictionary contents) that is immediately translated into a first cut logical schema. On the contrary, each program includes only a partial schema of standard files. At first glance, standard files are more tightly coupled with the programs and there are more structures and constraints buried into the programs. Unfortunately, all the standard file tricks have been found in recent applications that use "real" databases as well. The reasons can be numerous: to meet other requirements such as reusability, genericity, simplicity, efficiency; poor programming practice; the application is a straightforward translation of a file-based legacy system; etc. So, in both old and recent data-oriented applications, many data structures and constraints are not explicitly declared but are coded, among others, as the procedural sections of the program. For all these reasons, one of the data structure extraction source of information is the program text source.

The analysis of the program source is a complex and tedious task. This is due to the fact that procedurally-coded data constructs are spread in a huge amount of source files, and also because there is no standard way to code a definite structure or constraint. Each programmer has his personal way(s) to express them. This also depends on the programming language, the target DBMS, the enterprise rules, the programmer skill, his mood, etc.

We do not need to retrieve the complete specification of the program, we are merely looking for information that are relevant to find the undeclared structures of persistent data. More precisely, we are looking for evidences of fields refinement and aggregation, referential constraints and exact cardinalities, to mention only a few.

We will give some informal information on how the main implicit constructs can be recovered through program analysis.

- *Field refinement* consists in associating, with a field (or a record) initially declared atomic, a new structure comprising sub-fields. *Field aggregation* is the opposite operation, according to which a collection of independent fields appear to be the

components of a compound field, or the values of a multivalued field. These two constructions can be detected if there is a relation (assignment, comparison) between two variables of different structures.

- A *foreign key* consists of a field of a record that references another record. To detect a foreign key, we have to find evidences that each value of the source field belongs to the value set of the key of another record.
- The *cardinality* of an array states, through the couple I-J, that the number of active elements of the array is between I and J. The exact minimum cardinality (I) of an array can be found by the analysis of the array initialization. The maximum cardinality of an array is usually the size of the array, except if it can be found that the last elements are never used.

Several industrial projects have proved that powerful program understanding techniques and tools are essential to support the data structure extraction process in realistic size database reverse-engineering projects.

4. Program understanding techniques

In this section, we will present program understanding techniques that we use during the data structure extraction. The main techniques explained here are pattern matching, dependency graph, program slicing and program visualisation. For each of these techniques it will be shown how they can be applied to structure elicitation in database reverse engineering.

4.1 Search in text sources

The simplest way to find some definite information in a program is to search the program source text for some patterns or *clichés*. We use the term pattern and not just string, as in a text editor, because a pattern describes a set of possible strings. It can include wildcards, characters ranges, multiple structures, variables and can be based on other defined patterns. For example, we can write a pattern that match any numeric constant, or the various kind of COBOL assignment statements, or some `select-from-where` SQL queries. The pattern matching tool can be coupled with some external procedure that are triggered for each pattern match.

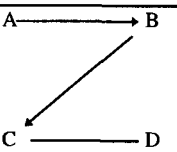
4.2 Dependency graph

The *dependency graph* (generalisation of dataflow diagram) is a graph where each variable of a program is represented by a node and an arc (directed or not) represents a direct relation (assignment, comparison, etc.) between two variables. Fig. 3.b illustrates the dependency graph of the program fragment of Fig. 3.a. If there is a path from variable A to variable C in the graph, then there is, in the program, a sequence of statements such that the value of A is in relation with the value of C. The very meaning of this relation between the variables is defined by the analyst: the

structure of one variable is included into the structure of the other one, the variables share the same values, they denote the same real world object, etc.

```
     ...
MOVE A TO B
     ...
MOVE B TO C
     ...
IF (C = D )
```

a) Fragment of a COBOL source



b) The dependency graph of the fragment

Fig. 3. The dependency graph.

The dependency graph will be used to find evidences for field refinement, field aggregation or foreign keys.

4.2 Program slicing

The slice of a program with respect to program point p and variable x consists of all the program statements and predicates that might affect the value x at point p. This concept was originally discussed by M. Weiser [10]. Since then, various slightly different notions of program slices have been proposed, as well as a number of methods to compute them.

```
   FD CUSTOMER.
      01 CUS.
        02 CUS-NUM PIC 9(3).
        02 CUS-NAME PIC X(10).
        02 CUS-ORD PIC 9(2).
           OCCURS 10.
        ...
      01 ORDER PIC 9(3).
        ...
1     ACCEPT CUS-NUM.
2     READ CUS KEY IS CUS-NUM.
3     MOVE 1 TO IND.
4     MOVE 0 TO ORDER.
5     PERFORM UNTIL IND=10
6       ADD CUS-ORD(IND) TO ORDER
7       ADD 1 TO IND.
8     DISPLAY CUS-NAME.
9     DISPLAY ORDER.
```

a) COBOL program P

```
   FD CUSTOMER.
      01 CUS.
        02 CUS-NUM PIC 9(3).
        02 CUS-NAME PIC X(10).

1     ACCEPT CUS-NUM.
2     READ CUS KEY IS CUS-NUM.
8     DISPLAY CUS-NAME.
```

b) Slice of P with respect to *CUS-NAME* and *line 8*

Fig.4. Example of program slicing.

Fig. 4.a is a small COBOL program that asks for a customer number (CUS-NUM) and displays the name of the customer (CUS-NAME) and the total amount of its order (ORDER). Fig. 4.b shows the slice that contains all the statements that contribute to displaying the name of the customer, that is the *slice of P with respect to CUS-NAME at line 8.*

Procedures, arbitrary control flow (Go To) and composite data types, typical to COBOL programs, make computing program slices a fairly complicated task. Our approach is based on the System Dependence Graph techniques developed by [6].

In DBRE, we use program slicing to reduce the scope of program analysis, either by visual inspection, or through the use of other tools. It is mainly used to find evidences of foreign keys, exact cardinality, local identifiers and domain constraints. The slice must be computed with respect to a statement that uses the candidate field. For example, a slice can be computed for an output statement and a candidate foreign key to see if each value of the foreign key is checked to be in the value set of the identifier of the target record.

4.3 Program representation

Program source code, even reduced through program slicing, often is too difficult to understand because the program can be huge (thousands of pages of code), poorly structured, based on poor naming conventions, and made up of hundreds of modules. To help the analyst grasp the structure of a program it is useful to give him different synthetic abstract views of it such as the *call graph* or the *input/output dataflow* of the different modules.

There are many other visualisations of a program that can be useful to help understand a program. The dependency graph can also be displayed as a graph. It is why we need a tool that can easily create new kinds of graph.

5. The tools provided by DB-MAIN

Program understanding is complex and tedious. Although the process cannot be fully automated, the analyst cannot carry it out without a powerful and flexible set of tools. These tools must be integrated and their results recorded in a common repository. In addition, the tools need to be easily extensible and customisable to fit the analyst's exact needs.

DB-MAIN is a general database engineering CASE environment that offers sophisticated reverse engineering toolsets. DB-MAIN is one of the results of a R&D project started in 1993 by the database team of the computer science department of the University of Namur (Belgium). Its purpose is to help the analyst in the design, reverse engineering, migration, maintenance and evolution of databases.

DB-MAIN offers the usual CASE functions, such as the database schema creation, management, visualisation, validation, transformation, and code and report generation. It also includes a programming language (*Voyager2*) that can manipulate the objects of the repository and allows the user to develop its own functions. More details can be found in [3] and [5].

DB-MAIN also offers some functions that are specific to DBRE. The *extractors* extract automatically the data structures declared into a source text. Extractors read the declaration part of the source text and create corresponding abstractions in the repository. The *foreign key assistant* is used to find the possible foreign keys of a schema. Giving a group of fields, that is the origin (or the target) of a foreign key, it searches a schema for all the groups of fields that can be the target (or the origin) of

the first group. The search is based on a combination of matching criteria such as the group type, the length, the type and matching rules of the name of the fields.

Other reverse engineering functions use program understanding processors.

A *pattern matching* engine searches a source text for a definite pattern. Patterns are defined into a pattern description language (PDL), they can use the definition of other patterns and contain variables. Those variables can be used as parameters of a procedure to be executed for each pattern instance.

```
var ::= /g"[a-zA-Z0-9]*[a-zA-Z][-a-zA-Z0-9]*";
var_1 ::= var;
var_2 ::= var;
move ::= "MOVE" - @var_1 - "TO" - @var_2 ;
```

Fig. 5. The COBOL `move` pattern definition.

Fig. 5 shows the definition of the `move` pattern, this pattern will match with the simplest form of COBOL assignments. The PDL variables are prefixed with the '@' character and the '-' name denotes a pattern (inter-token separators) defined in a secondary library.

DB-MAIN offers a *variable dependency graph* tool. The user defines the relations between variables as a list of patterns. Each pattern must contain exactly two variables and the instances of those variables are the nodes of the graph. For example, if the pattern `move` (Fig. 6) is given then the arcs of the dependency graph will represent variables connected by a assignment statement. The dependency graph itself is displayed *in context*: the user selects a variable, then all the occurrences of this variable, and of all the variables connected to it in the dependency graph are coloured into the source text, both in the declaration and in the procedural sections. Though a graphical presentation could be thought to be more elegant and more abstract, the experience has taught us that the source code itself gives much *lateral* information, such as comments, layout and surrounding statements.

The *program slicing* tool computes the program slice with respect to the selected line of the source text and one of the variables, or component thereof, referenced at that line. When the slice is computed its statements are coloured, so that it can be examined in context. For example, some lines that do not belong to the slice (e.g., comments or error messages) may give some additional hints to understand the slice. If needed, the slice can be transformed into a new program to be processed by other analyzers.

6. Conclusion

Though the *software engineering* and *database/IS* communities most often live on different planets, experience proves that many real world problems cannot be solved without combining the concepts, techniques and reasonings belonging to each of them. Anyway, an information system mainly comprises a database and a collection of programs. Therefore, understanding these programs needs a deep understanding of

the semantics of the database, while recovering this semantics cannot ignore what the programs are intended to do on/with the contents of the database.

One of the major objectives of the DB-MAIN project is the methodological and tool support for database reverse engineering processes. We have quickly learned that we needed powerful program analysis reasoning and their supporting tools, such as those that have been developed in the program understanding realm. We integrated these reasoning in a highly generic DBRE methodology, while we developed specific analyzers to include in the DB-MAIN CASE tool.

One of the great lessons we painfully learned is that they are no two similar DBRE projects. Hence the need for easily programmable, extensible and customizable tools. The DB-MAIN (meta-)CASE tool is now a mature environment that offers powerful program understanding tools dedicated, among others, to database reverse engineering, as well as sophisticated features to extend its repository and its functions. DB-MAIN has been developed in C++ for MS-Windows workstations. An education version is available at no charge for non-profit institutions (http://www.info.fundp.ac.be/~dbm).

References

1. Anderson, M.: Reverse Engineering of Legacy Systems: From Valued-Based to Object-Based Models, *PhD thesis*, Lausanne, EPFL (1997)
2. Batini, C., Ceri, S. and Navathe, S.B.: Conceptual Database Design - An Entity-Relationship Approach, Benjamin/Cummings (1992).
3. Englebert, V., Henrard J., Hick, J.-M., Roland, D. and Hainaut, J.-L.: DB-MAIN: un Atelier d'Ingénierie de Bases de Données, in *11 Journée BD Avancées*, Nancy (1995).
4. Hainaut, J.-L., Chandelon, M., Tonneau, C. and Joris M.: Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press (1993).
5. Hainaut, J.-L, Roland, D., Hick J-M., Henrard, J. and Englebert, V.: Database Reverse Engineering: from Requirements to CARE Tools, *Journal of Automated Software Engineering*, **3**(1) (1996).
6. Horwitz, S., Reps, T. and Binkley, D. Interprocedural Slicing using Dependence Graphs, ACM Trans. on Programming Languages and Systems **12**(1), Jan. 1990, 26-60 (1990).
7. Joris, M., Van Hoe, R., Hainaut, J.-L., Chandelon, M., Tonneau, C. and Bodart, F. et al.: PHENIX: Methods and Tools for Database Reverse Engineering, in *Proc 5th Int. Conf. on Sofwazre Engieering and Applications.* Toulouse, EC2 Publish (1992).
8. Petit, J.-M., Kouloumdjian, J., Bouliaut, J.-F. and Toumani, F.: Using Queries to Improve Database Reverse Engineering, in *Proc of the 13th Int. Conf. on ER Approach*, Manchester. Springer-Verlag (1994).
9. Rugaber S. Program Comprehension. Technical report, College of Computing, Georgia Institute of Technology (1995).
10. Weiser, M.: Program Slicing, *IEEE TSE*, **10**, 352-357 (1984).