Data structure extraction in database reverse engineering

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

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

Abstract. Database reverse engineering is a complex activity that can be modeled as a sequence of two major processes, namely data structure extraction and data structure conceptualization. The first process consists in reconstructing the logical - that is, DBMS-dependent - schema, while the second process derives the conceptual specification of the data from this logical schema. This paper concentrates on the first process, and more particularly on the reasonings and the decision process through which the implicit and hidden data structures and constraints are elicited from various sources.

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.

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 database 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 may be 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 assume that all the conceptual specifications have been declared into the DDL (data description language), so that all the other information sources are ignored. In addition, the schema has not been deeply restructured for performance or for any other requirements, and names have been chosen rationally.

These conditions cannot be assumed for most large operational databases. Since the early nineties, some authors have recognised that the analysis of the other sources of information is essential to retrieve data structures ([1], [4], [8], [9] and [10]).

The constructs that have been declared in the DDL are called *explicit* constructs. On the opposite, the constraints and structures that have not been declared explicitly are called *implicit* constructs. The analysis of DDL statements alone leaves the implicit construct undetected.

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 contents, etc.) can accumulate evidences for those specifications.

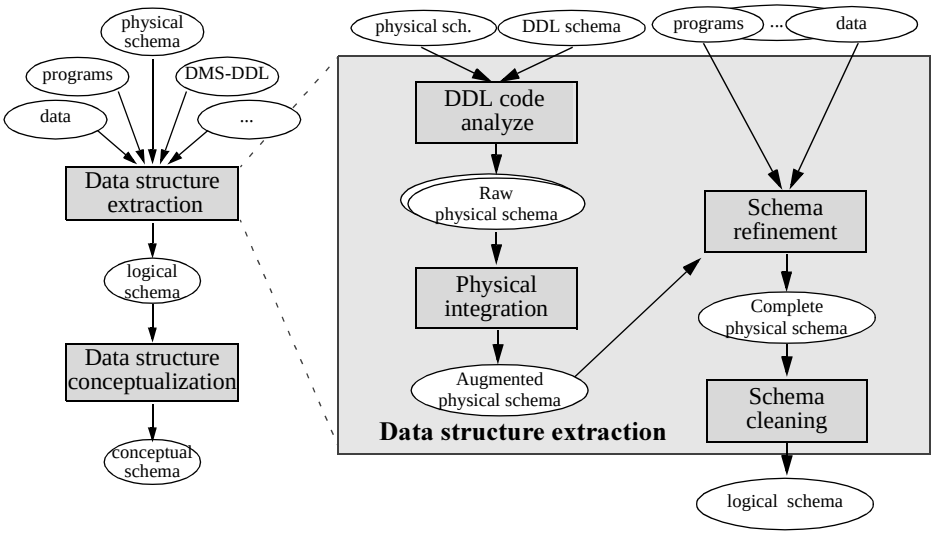

Fig. 1. The major processes of the reference DBRE methodology (left) and the development of the data structure extraction process (right).

This paper presents in detail the refinement process of our database reverse engineering methodology. This process extracts all the implicit constructs through the analyze of the information sources. The paper is organized as follows. Section 2 is a synthesis of a generic DBMS-independent DBRE methodology. Section 3 describes the data structure extraction process. Section 4 discus how to refine a schema to enrich it with all the implicit constraints. Section 5 presents a DBRE CASE tool which is intended to support data structure extraction, including schema refinement.

2 A Generic Methodology for Database Reverse Engineering

The reference DBRE methodology [4] is divided into two major processes, namely *data structure extraction* and *data structure conceptualization* (Fig. 1, left). These problems address the recovery of two different schemas and require different concepts, reasoning and tools. In addition, they grossly appear as the reverse of the physical and logical design usually admitted in database design methodologies [2].

2.1 Data Structure Extraction

The first process consists in recovering the complete DMS schema, including all the explicit and implicit structures and constraints, called the *logical schema.*

It is interesting to note that this schema is the document the programmer must consult to fully understand all the properties of the data structures (s)he intends to work on. In some cases, merely recovering this schema is the main objective of the programmer, who can be uninterested in the conceptual schema itself.

This process will be discussed in detail in section 3 and 4.

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 optimization and DMS-dependent constructs.

The final product of this phase is the conceptual schema of the persistent data of the application. More detail can be found in [5].

2.3 An Example

Fig. 2 gives a DBRE process example. The files and record declarations (2.a) are analyzed to yield the physical schema (2.b). This schema is refined through the analysis of the procedural parts of the program (2.c) to produce the logical schema (2.d). This schema exhibits two new constructs, namely the refinement of the field CUS-DESC and the foreign key. It is then transformed into the conceptual schema (2.e).

3 Data Structure Extraction

The goal of this phase is to recover the complete DMS schema, including all the implicit and explicit structures and constraints. The main problem of the data structure extraction is to discover and to make explicit, through the refinement process, the structures and constraints that were either implicitly implemented or merely discarded during the development process. In the reference methodology we are discussing, the main processes of data structure extraction are the following (Fig. 1, right):

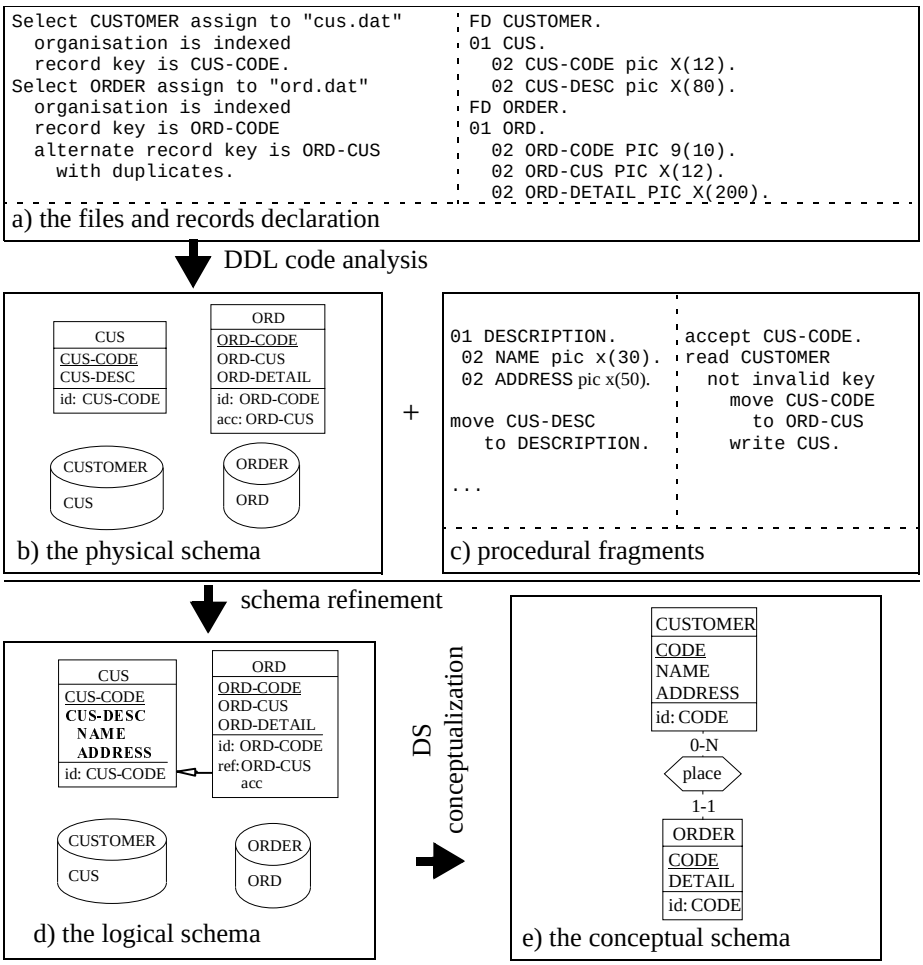

Fig. 2. Database reverse engineering example.

- *DDL code analysis*: analysing the data structure declaration statements to extract the explicit constructs and constraints, thus providing a *raw physical schema*. This is one of the simplest process in DBRE which can be automated easily (most database-oriented CASE tools provide a collection of such analyzers).
- *Physical integration*: when more than one DDL source has been processed, several extracted schemas can be available. All these schemas are integrated into one global schema. The resulting schema (*augmented physical schema*) must include the specifications of all these partial views.
- *Schema refinement*: the explicit physical schema obtained so far is enriched with implicit constructs made explicit, thus providing the *complete physical schema.* This process will be discused in section 4.

- *Schema cleaning*: once all the implicit constructs have been elicited, technical constructs such as indexes or clusters are no longer needed and can be discarded in order to get the *complete logical schema* (or simply the logical schema).

The final product of this phase is the complete logical schema, that includes both explicit and implicit structures and constraints. This schema is no longer DMS-compliant for at least two reasons. Firstly, it is the result of the integration of different physical schemas, which can belong to different DMS. For example, some part of the data can be stored into a relational DB, while others are stored into standard files. Secondly, the complete logical schema is the result of the refinement process, that enhances the schema with recovered implicit constraints.

The data structure extraction process is often easier for true database than for standard files. Indeed, databases have a global schema (DDL text) that is immediately translated into a physical 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" database 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.

Some kind of integration can also be necessary later to integrate different components of the schema that are represented by different structures (different record types for example) but represent the same concept. This can be discovered during the schema refinement process that will be presented in the next section.

4 Schema refinement

The main problem of the data structure extraction phase is to discover and to make explicit, through the refinement process, the structures and constraints that were either implictly implemented or merely discarded during the development process. The variety of implicit constructs can be very large, the main implicits structures and constraints we are looking for are the following: record types and fields desaggregation, identifier, foreign keys, functional dependency, meaningful names, etc.

In this section, we analyse why we need different sources of information, we describe the elicitation techniques, then we present a generic refinement methodology. We conclude by the analysis of the automatization of the refinement process.

4.1 The Information Sources

To discover an implicit construct the analyst cannot limit its analysis to one information source. On the contrary (s)he has to rely on all the possible information sources. Those sources are for example: application programs, data, HMI procedural fragments, screen and report layout, generic DMS code fragments[1], existing documentation, interviews, domain knowledge, operation environment knowledge, etc.

(S)he needs to analyze several of those sources because none of them contains all the hints for all the constraints. For example, some constraints are not implemented in the application program because they are verified by some environmental properties (the input data are always correct, they come from another fully reliable application). On the other hand, spurious constraints can be discovered in the data (for example, a field is an identifier) because the set of data is too small. Or constraints are not verified because there is some erroneous data.

Many data structures and constraints, that are not explicitly declared, are coded, among other, as procedural sections of the programs. For this reason, one of the most important information source is the program text sources.

4.2 The Refinement Techniques

For each information source there exists a set of analysis techniques. These techniques can be very simple such as visual inspection of the data or more complex such as dynamic analysis of executing programs.

```
   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. 3. Example of program slicing.

Program understanding is a very active area within the software engineering field. It is the process of acquiring knowledge about an existing computer program [6]. The main program understanding techniques are:

- searching the program source text for some patterns or clichés;
- dependency graph: the dependency graph is a graph where each variable of a program is represented by a node and an arc represents a direct relation (assignment, comparison, etc.) between two variables;
- 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 [11]. Fig. 3.a is a

1 Some DMS offer general functionality to enforce a large variety of constraints on the data.

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. 3.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.*

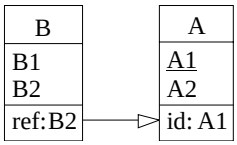

Fig. 4. An elementary abstract schema including a foreign key. A1 is a identifier of table A and B2 is a foreign key targeting table A.

Sometimes it can be useful to have different visualization techniques for the same information. A program can be visualized as a text or as a call graph.

Data can be analyzed through queries that verify whether a constraint is present or not. For example, it is possible to write a query to verify that, in Fig. 4, B2 is a foreign key targeting table A:

```
select count(*) from B where B2 not in (select A1 from A);
```

If the result is 0, then B2 could be a foreign key, otherwise it cannot.

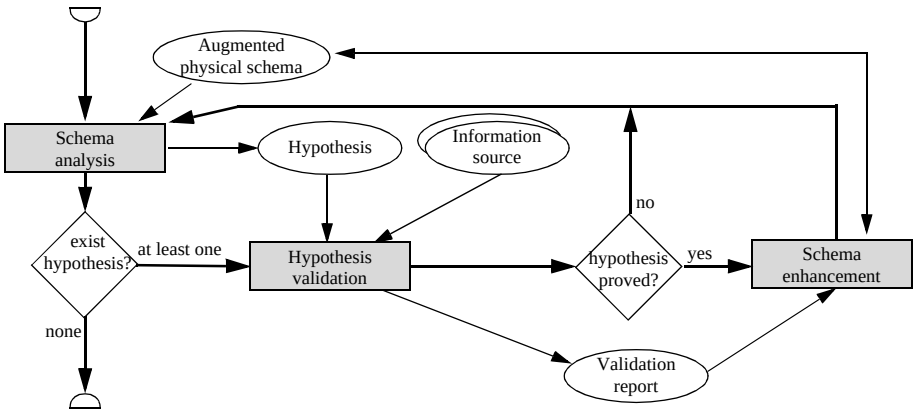

Fig. 5. The refinement method.

4.3 Schema Refinement Method

Due to the large amount of information to manipulate, attempting an exhaustive search for all the imaginable constraints is unrealistic. We need a methodology that guide us in our constraints investigation. This methodology will reduce the search space to the possible constraints. For example, it is not realistic to query the data to check for each field combination of the database being an identifier. We have to decide which fields are potential identifiers depending of their name (containing key word as “code”, “id”, “number”, etc.), their structure (mandatory), their position in the record (the first field of the record), etc.

Fig. 5 sketches a schema refinement method, where rectangles represent process, ellipses represent the different products (schema, data passed from one process to the other, ...), diamonts shapes represent decision points. The execution flow is materialized by bold arrows and normal arrows represent product usage. The *schema analysis* process analyses the *Augmented physical schema* to find missing constraints or constructs, called *hypothesis*. The domain knowledge and the database design knowledge are also used to discover missing constraints during the schema analysis. Then we analyse the other sources of information to validate the hypothesis (*hypothesis validation*). If the hypothesis is validated then the *schema enhancement* process enriches the schema with the validated constraint. We iterate until no new hypothesis are generated by the schema analysis.

This method can be instantiated for each DBRE project. Schema analysis depends on the constraints we are looking for and hypothesis validation change according to the information source and the constraint we want to validate. Indeed, database reverse engineering basically is a loosely structured learning process which varies largely from one project to another. We can sketch, as an example, the following principles for foreign key elicitation that can apply on relational databases managed by early RDBMS, in which no keys were explicitly declared. This example is based on the schema of the Fig. 4.

Phase	Heuristics	Short description
Schema analysis	name analysis	Name of column B.B2 suggests a table, or an external id, or includes keywords such as ref, ...
	name analysis	Select table A based on the name of B2
	domain knowledge	Objects described by B are known to have some relation with those described by A
	domain knowledge	Find a table describing objects which are known to have some relation with those described by B
	technical constructs	Search the schema for a candidate referenced table and id (with same type and length)
Hypothesis	*B.B2 >> A.A1*	*field B2 of B is a foreign key to identifier A1 of A*
Hypothesis Proving	technical constructs	There is an index on B2
	technical constructs	B.2 and A.A1 are in the same cluster
	dataflow analysis	B.2 and A.A1 are in the same dataflow graph fragment
	usage pattern	A.A1 values are used to select B rows with same B2 values
	usage pattern	B.2 values are used to select A row with same A1 value
	usage pattern	A B row is stored only if there is a matching A row
	usage pattern	When an A row is deleted, B rows with B2 values equal to A.A1 are deleted as well
	usage pattern	There are views based on a join with B.B2 = A.A1
Hypothesis Disproving	data analysis	Prove that some B.B2 values are not in A.A1 value set

It is a good practice to apply as many heuristics as we can. Because if an heuristic succeeds, it does not mean that the hypothesis is verified. On the opposite, if a heuris-

tics fails, the hypothesis is not necessarily disproved. We can formalize this as follows:

- we formulate hypothesis *h* on the existence of an implicit construct *C*;
  so far, *h* is stated with probability $p_0 < 1$;
- we apply heuristics *H*; *h* is now stated with probability p_1:
  - if *H* succeeds
    $p_1 > p_0$; the existence of *C* is more certain, though $p_1 < 1$.
    For instance, in the example above, if there is an index on B2 it is one more evidence that B2 is a foreign key to the identifier A1 of A, but we are not yet completely certain.
  - if *H* fails, one the three interpretations can hold:
    - $p_1 = 0$; *h* is disproved, so that we accept that *C* does not exist.
      For example, if half of the value of B.B2 are not in A.A1 value set, we can say that there is no foreign key from B.B2 to A.A1.
    - $p_1 < p_0$; *h* is less certain, but could still be proved through other heuristics.
      For example, if there is only one value of B.B2 (out of one million) that is not in A.A1 value set, we can not conclude there is no foreign key, but it is perhaps an error in the data.
    - *H* does not contribute to the search; $p_1 = p_0$.
      For example, if we didn't find that B.B2 and A.A1 are in the same dataflow graph fragment. This can be because we have chosen a program that does not use the foreign key (a program that only manipulate B and not A).

The experience has shown that:

- Analysing all the information sources generally proves too expensive, so that the analyst has to determine which sources to analyse.
- A hypothesis cannot be proved by heuristics alone, it is up to the analyst to decide when (s)he is convinced the hypothesis is validated.
- This method considers that all the information are reliable, what about the result of an heuristic applyied on unreliable information (corrupted data, programming errors)?
- When we are validating an hypothesis, we can discovered other constraints that must be added to the schema. For example, when analyzing a program slice computed to verify that a field is a foreign key, other constraints about this field can be found, because the slice contains all the program instructions that influence the value of the field.

4.4 How to Decide that Refinement is Completed

The goal of the reverse engineering process has a great influence on the output of the refinement process and on its ending condition. The simplest DBRE project can be to recover only the list of all the records with their fields. All the other constraints are useless. This can be useful to make a first inventory of the data used by an application

to prepare another reverse or maintenance project (as Y2K conversion). In this kind of project, the logical schema is the final product of DBRE.

On the other end, we can try to recover all the possible constraints to have a complete view of the database. This can be necessary in a migration project where we want to convert a collection of standard files into a relational database.
It is the analyst's responsibility to decide that the schema is complete and all the needed constraints have been extracted.

4.5 Process Automation

The schema refinement process basically is a decisional activity that cannot be fully automated. Many analysis techniques are not intended to locate and find implicit constructs, but rather contribute to the discovery of these constructs by focusing the analyst's attention on special text or structural patterns. In short, they narrow the search scope. It is up to the analyst to decide if the constraint that is looked for is present or not. For example, computing a program slice provides a small set of statements with a *high density of interesting patterns* according to the construct that is searched for (typically a foreign key). This small program segment must then be examined visually to check whether traces of the construct are present or not.

Another reason for which full automation cannot be reached is that each DBRE project is different. The source of information, the underling DBMS or the coding rules, can all be different and even incompatible.

Despite these restrictions, automation is highly desirable for large projects in which huge volumes of information have to be explored. Portfolios of more than 10,000 programs and databases of more than 500 files/tables are not unfrequent[1].

This automation can be of different kinds:

- Some processes can be fully automated. For example, during the schema analysis process, it is possible to have a tool that detects all the possible foreign key that meet some matching rules (the target is an identifier and the candidate foreign keys have the same length and type as their target).
- Other processes can be partially automated with some interaction with the analyst. For example, we can use the dataflow diagram to detect automatically the fields decomposition. There is an interaction with the analyst to resolve conflicts (two different decomposition for the same fields).
- We can define tools that generate reports so that the analyst can analyse them to validate the existence of a constraint. For example, we can generate a report with all the fields that contain the key words "id", "code" and are the first field of their record. The analyst must decide which fields are candidate identifiers.

---

1 The complexity of DBRE projects is between O(N) and O(N^2), where N is the number of entity types of the schema. Indeed, each new entity type can be related with all the entity types of the schema. The analysis effort to process a 500-table database can be up to 100 times greater than for a 50-table database

5 The DBRE Functions of DB-MAIN

Several industrial projects have proved that powerful techniques and tools are essential to support DBRE, especially the data structure extraction process, in realistic size projects. These tools must be integrated and their results recorded in a common repository. In addition, the tools need to be easily extensible and customizable to fit the analyst's exact needs.

DB-MAIN is a general-purpose 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 database applications.

DB-MAIN offers the usual CASE functions, such as 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 detail can be found in [3] and [5].

DB-MAIN also offers some functions that are specific to the data structure extraction process [6]. 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 the name of the constructs.

Other reverse engineering functions use three specific program understanding processors.

- A *pattern matching* engine searches a source text for a definite pattern. Patterns are defined into a powerful pattern description language (PDL), through which hierarchical patterns can be defined.
- DB-MAIN offers a *variable dependency graph* tool. 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.

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.

6 Conclusion

In this paper, we have presented a generic methodology for the data structures extraction process. We have shown that the schema refinement is a difficult task because it cannot be fully automated and it can be very different from one project to another.

The role of the analyst is very important. Except in simple projects, (s)he needs to be a skilled person, who is competent in the application domain, in database design methodology, in DBMS's and in programming language (usually old ones).

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.

An education version is available at no charge for non-profit institutions (http://www.info.fundp.ac.be/~dbm).

7 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, *Ingénierie des Système d'Information*, V4 n°1, HERMES-AFCET (1996).
4. Hainaut, J.-L., Chandelon, M., Tonneau, C. and Joris M.: Contribution to a Theory of Database Reverse Engineering, in *Proc. of WCRE'93*, Baltimore, 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. Henrard, J., Englebert, V., Hick, J-M. , Roland, D. , Hainaut, J-L.: Program understanding in databases reverse engineering, in *Proc. of DEXA'98*, Vienna (1998).
7. Jerding, D., Rugaber, S.: Using Visualization for Architectural Localization and Extraction, in *Proc. of WCRE'97*, Amsterdam (1997).
8. 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 Software Engieering and Applications.* Toulouse, EC2 Publish (1992).
9. 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).
10. Montes de Oca C., Carver D. L., A Visual Representation Model for Software Subsystem Decomposition, in *Proc of WCRE'98*, Hawai, USA, IEEE Computer Society Press (1998).
11. Weiser, M.: Program Slicing, IEEE TSE, 10, 352-357 (1984).