Strategies for Data Reengineering

Jean Henrard, Jean-Marc Hick, Philippe Thiran, Jean-Luc Hainaut
Database Applications Engineering Laboratory
Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur - Belgium
{jhe, jmh, pth, jlh}@info.fundp.ac.be

Abstract

This paper describes and analyzes a serie of strategies to migrate data-intensive applications from a legacy data management system to a modern DMS. Considering two ways to migrate the data and three ways to propagate the corresponding perturbation to the program code, the paper identifies six reference strategies that provide different levels of quality and induce different costs. Three of them are discussed in detail and illustrated by the conversion of COBOL files into a SQL database.

1. Introduction

Brodie and Stonebraker [4] define legacy information system (IS) as large and old programs built around legacy database (IMS, CODASYL, ...) or using primitive DMS[1] (like COBOL file system, ISAM, ...). Legacy IS are also independent in that they do not interface with other applications. Moreover, they have proved critical to the business of organizations. To keep being competitive, organizations must improve their IS and invest in advanced technologies. In this context, the claimed 50-80 percent cost of legacy systems maintenance (w.r.t. total cost) is considered prohibitive [18].

A popular solution of the legacy systems evolution problems is the migration, i.e. moving the applications and the databases to new platform and technologies. Migration also is an expensive and complex process. But it greatly increases the IS control and evolution to meet future business requirements. The scientific literature ([3], [4]) mainly identifies two migration strategies: rewriting the legacy IS from scratch or migrating by small incremental steps. The incremental strategy allows the migration projects to be more controllable and predictable in terms of deadline and budget. The difficulty lies in the determination of the migration steps.

1 Data Management System.

Legacy system migration is concerned with implementing a new system that preserves the functionalities and data of the original system. The data assets stored in a database are a critical part of legacy systems. A modern DMS can increase data control, performance and independence. In this context, an important step into the incremental strategy commonly is the data (structures and instances) migration.

To make the discussion more concrete, we base it on the most popular problem pattern, that is, the conversion of a legacy COBOL program, using standard indexed files, into an equivalent COBOL program working on a relational database. The discussion is to a large extend valid for other languages and other DMS. This paper presents a practical approach to data-intensive application reengineering based on two independent dimensions. The first one relates to the quality of the target database and the second one describes the way the application programs are altered, so that six strategies can be derived. We will describe and discuss three of them.

Legacy IS migration is a major research domain. Some general migration methods are available. For example, [16] discusses current issues and trends in legacy system reengineering from several perspectives (engineering, system, software, managerial, evolutionary, and maintenance). The authors propose a framework to place reengineering in the context of evolutionary systems. The butterfly methodology [19] provides a migration methodology and a generic toolkit for the methodology to aid engineers in the process of migrating legacy systems. Different from the incremental strategy, this methodology eliminates the need of interoperability between the legacy and target systems. Renaissance is an ongoing ESPRIT project that develops a method for system evolution and reengineering. It provides technical guidelines [14] for the migration of legacy systems to distributed client/server architectures.

Closer to our data-centered approach, the Varlet project [6] adopts a process that consists in two phases. In the first one, the different parts of the original database are analyzed to obtain a logical schema for the implemented physical schema. In the second phase, this logical schema is transformed into a conceptual one, which is the basis for

modification or migration activities. The approach of Jeusfeld [7] is divided into three parts: mapping of the original schema into a meta model, rearrangement of the intermediate representation and production of the target schema.

Some works also address the migration between two specific systems. Among those, Menhoudj and Ou-Halima [12] present a method to migrate the data of legacy system into a relational database management system. Behm and al. [2] describe a general approach to migrate relational database to object technology.

The reminder of the paper is organized as following. Section 2 specifies the problem, describes six strategies of data reengineering and a CASE support. Section 3 presents an environment to support the migration process. Section 4 develops the data migration strategies. Section 5 analyzes the programs modifications generated by the data reengineering and section 6 compares the analyzed strategies and concludes this paper.

2. Problem statement

Data reengineering consists in deriving a new database from a legacy database and in adapting the software components accordingly [4]. It comprises three main steps: (1) schema conversion, (2) data conversion and (3) program conversion.

Schema conversion is the translation of the legacy database structure, or schema, into an equivalent database structure expressed in the new technology. Both schemas must convey the same semantics, i.e., all the source data should be losslessly stored into the target database[2]. Most generally, the transformation of a source schema to a target schema is made up of two processes. The first one, called database reverse engineering (DBRE) [8], aims to recover the conceptual schema that expresses the semantics of the source data structure. The second process is standard and consists in deriving the target physical schema from this conceptual specification.

Data conversion is the migration of the data instance from the legacy database to the new one. This migration involves data transformations that materialize the schema transformations described above.

Program conversion, in the context of data reengineering, is the modification of the program so that it now accesses the migrated database instead of the legacy data. The functionalities of the program are left unchanged, as well as its programming language and its user interface (they can migrate too, but this is another problem). Program conversion can be a complex process in that it relies on the rules used to transform the legacy schema into the target schema.

2.1. Strategies

As already mentioned, we consider two dimensions, namely the database conversion and the program conversion.

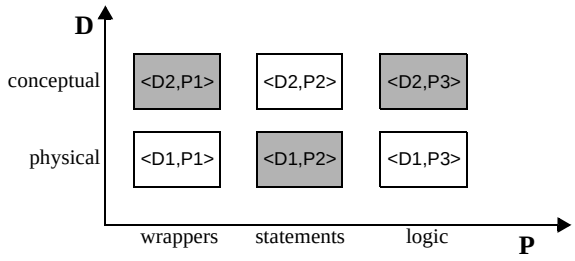


Figure 1. The six reference IS migration strategies.

The database dimension. We consider two extreme database conversion strategies leading to different levels of quality of the transformed database. The first strategy (*Physical conversion* or D1) consists in translating each construct of the source database into the closest constructs of the target DMS without attempting any semantic interpretation. The process is quite cheap, but it leads to poor quality databases with no added value. The second strategy (*Conceptual conversion* or D2) consists in recovering the precise semantic description (i.e., its conceptual schema) of the source database first, through reverse engineering techniques, then in developing the target database from this schema through a standard database methodology. The target database is of high quality according to the new DMS expressiveness and is fully documented, but the process is more expensive.

The program dimension. Once the database has been converted, several approaches to application programs modification can be followed. We identify three strategies. The first one (*wrappers* or P1) relies on *wrappers* that encapsulate the new database to provide the application programs with the legacy data access logic, so that these programs keep reading and writing records in (now fictive) indexed files, possibly through program calls instead of through native file statements. The second strategy (*Statement rewriting* or P2) consists in rewriting the access statements in order to make them process the new data through the new DMS-DML[3]. For instance, a `READ` COBOL statement is replaced with a `select-from-where` (`SFW`) or a `fetch` SQL statement. In these two first strategies, the program logic is neither elicited nor changed. According to the

2 The concept of *semantics preservation* is more complex, but this definition is sufficient in this context.

3 DML: Data Manipulation Language.

third strategy (*Logic rewriting* or P3), the program is rewritten in order to use the new DMS-DML at its full power. It requires a deep understanding of the program logic, since the latter will generally be changed.
These dimensions define six information system migration strategies (Figure 1). Due to the limited scope of this paper, we will discuss three of them, namely <D2,P1>, <D1,P2> and <D2,P3>.

2.2. Case study

The following discussion is based on a case study in which the legacy system comprises a small COBOL program (300 lines of code) and three files. This program records and displays information about customers that place orders. The objective of the case study is to convert the legacy files into a new relational database and to transform the application program into a new COBOL program, with the same business functions, but that accesses the new database.

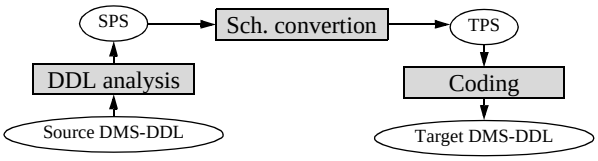
2.3. CASE support

Though this paper mainly focuses on strategy and methodology, we would like to suggest some requirements and to report on experience in IS conversion support. Real-size reengineering projects require powerful techniques and computer-aided tools. Such tools have been developed as customized plug-ins of DB-MAIN[4], a general-purpose database engineering CASE and meta-CASE environment that offers sophisticated engineering toolsets. Its purpose is to help the analyst in the design, reverse engineering, reengineering, maintenance and evolution of database applications. DB-MAIN includes several processors specific to the DBRE process [11], such as DDL extractors, a foreign key as*sistant*, and program analysis tools (a.o., for pattern matching, variable dependency analysis and program slicing).

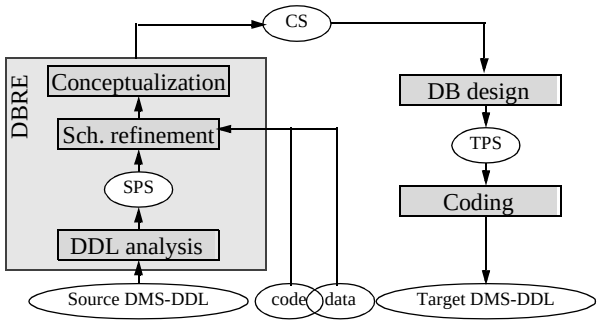
One thing we learnt is that two similar reengineering projects never exist. Hence the need for programmable, extensible and customizable tools. DB-MAIN (and more specifically its meta functions) includes features to extend its repository and its functions. In particular, it includes a 4GL (*Voyager2*) that allows analysts to develop their own customized processors [8].

3. Schema conversion

The schema conversion process analyzes the legacy application to extract the source physical schema of the underlying database (SPS) and transforms it into a target physical schema (TPS) for the target DMS. The TPS is used to generate the DDL code of the new database. In this section, we present two transformation strategies. The first strategy (Figure 2.a), the *Physical schema conversion*, merely simulates the structure of the legacy database into the target DMS. According to the second one (Figure 2.b), the *Conceptual schema conversion*, the complete semantics of the legacy database is retrieved and represented into the conceptual schema (CS). Then CS is used to develop the new database.



a) Physical schema conversion.



b) Conceptual schema conversion.

Figure 2. The two schema conversion strategies.

3.1. Physical conversion strategy (D1)

3.1.1. Principle

According to this strategy (Figure 2.a) each explicit data structure of the legacy database is directly translated into a similar structure of the target database (e.g., each record type becomes a table and each top-level field becomes a column). As opposed to the second strategy, no conceptual schema is built, so that the semantics of the data is ignored.

3.1.2. Methodology

The *DDL analysis* parses the DDL code to retrieve the physical schema of the legacy database (SPS). This schema includes all the data structures and constraints explicitly declared into the DDL code (Figure 2.a/left). This schema is then converted into its target DMS equivalent (TPS) through a straightforward *one-to-one* mapping and finally coded.

4 www.db-main.be.

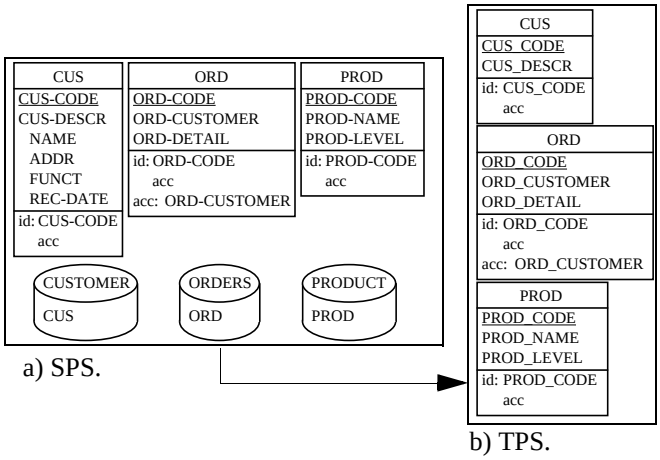


Figure 3. Example of Physical schema conversion.

3.1.3. Support

This strategy uses simple tools only, such as a DDL parser (to extract the SPS), an elementary schema converter (to transform the SPS into the TPS) and a DDL generator. Complex analyzers are not required.

3.1.4. Illustration

The analysis of the files and records declarations produces the SPS (Figure 3.a). Each COBOL record type is translated into a SQL table. The compound field `CUS-DESCR` cannot be expressed in SQL, so we transform it into an unstructured column with the same total length. Some names need to be changed to comply with the SQL syntax (Figure 3.b).

3.2. Conceptual conversion strategy (D2)

3.2.1. Principle

The physical schema of the legacy database (SPS) is extracted and transformed into a conceptual schema (CS), through a DBRE process. Then CS is transformed into the physical schema of the target system (TPS) through standard database development techniques.

3.2.2. Methodology

Figure 2.b/left depicts a simplified version of the methodology used to perform DBRE. A complete presentation of this methodology can be found in [8].

The *DDL analysis* parses the DDL code to retrieve the physical schema (SPS).

In the *schema refinement* process, the schema is refined through an in-depth inspection of the way the program uses and manages the data. Through this process, additional structures and constraints are identified, which were not explicitly declared but expressed in the procedural code. The existing data can also be analyzed, either to detect constraints, or to confirm or discard hypotheses on the existence of constraints.

The final DBRE step is the *data structure conceptualization* that interprets the physical schema into the conceptual schema (CS). Both schemas have the same semantics.

The physical schema of the new database (TPS) is derived from CS through standard database techniques (Figure 2.b/right). TPS is then used to generate the DDL of the target database.

3.2.3. Support

Extracting SPS and storing it in the CASE tool repository is done through a DDL extractor (SQL, COBOL, IMS, CODASYL, RPG) from the parser library.

Schema refinement requires schema and program analyzers as described in Section 2.3.

Data structure conceptualization and database design are based on schema transformation. Code generators produce the DDL code of the new database according to the specifications of TPS.

3.2.4. Illustration

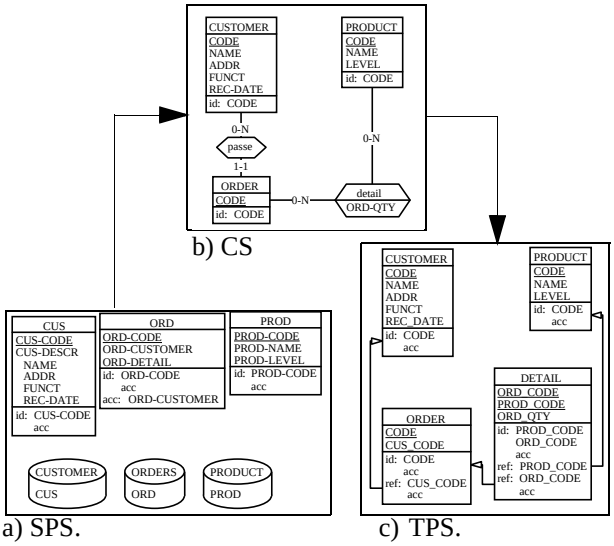


Figure 4. Example of Conceptual schema conversion.

The DDL code analysis reads the files and records declarations to retrieve the SPS (Figure 4.a).

Through the analysis of the variable dependency graph, or program slices and of the names and structure patterns of the schema, fields such as `ORD-DETAIL` are refined, foreign keys are discovered and constraints on multivalued fields are elicited. During the data structure conceptualization, the physical constructs (indexes, files) are removed and the implementation objects (record types, fields, foreign keys, arrays,...) are transformed into their conceptual equivalent (Figure 4.b). The detail of this DBRE case study has been described in [10].

The database design process transforms the entity types, the attributes and the relationship types into relational constructs such as tables, columns, keys and constraints. Finally physical constructs (indexes and storage spaces) are defined (Figure 4.c) and the code of the new database is generated.

3.3. Mapping definition

A careful analysis of the processes that have been described in Sections 3.1 and 3.2 shows that deriving a schema from another one is performed through techniques such as renaming, translating, conceptualizing, which basically are schema transformations. Most database engineering processes can be formalized as a chain of schema transformations as demonstrated in [9].

3.3.1. Schema transformation

Roughly speaking, a schema transformation consists in deriving a target schema `S'` from a source schema `S` by replacing construct `C` (possibly empty) in `S` with a new construct `C'` (possibly empty). Adding an attribute to an entity type, replacing a relationship type with an equivalent entity type or with a foreign key are three examples of schema transformations.

More formally, a transformation **`T`** is defined as a couple of mappings `<T,t>` such as: $c' = T(c)$ and $c' = t(c)$, where `c` is any instance of `C` and `c'` the corresponding instance of `C'`.

Structural mapping `T` explains how to modify the schema while *instance mapping* `t` states how to compute the instance set of `C'` from the instances of `C`.

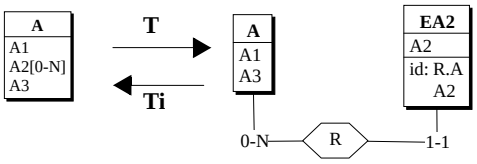


Figure 5. Representation of the structural mapping of a transformation that replaces attribute A2 with entity type EA2.

Any transformation **`T`** can be given an inverse transformation **`Ti`** = `<Ti,ti>` such as `Ti(T(C))=C`. If, in addition, we also have: `ti(t(c))=c`, then **`T`** (and **`Ti`**) are said semantics-preserving.

3.3.2. Compound transformation

A compound transformation **`T1 = T2 o T1`** is obtained by applying **`T2`** on the database that results from the application of **`T1`** [9].

An important conclusion of the transformation-based analysis of database engineering processes is that most of them, including reverse engineering and database design, can be modelled through semantics-preserving transformations. For instance, transforming `CS` into `TPS` can be modelled as a compound semantics-preserving transformation **`CS-to-TPS`** = `<CS-to-TPS, cs-to-tps>` in such a way that: `TPS = CS-to-TPS(CS)`. This transformation has an inverse: **`TPS-to-CS`** = `<TPS-to-CS,tps-to-cs>` such as: `CS = TPS-to-CS(TPS)`.

3.3.3. Transformation history

The *history* of an engineering process is the formal trace of its chain of transformations. In a history, a transformation is entirely specified by its signature, which specifies the name of the transformation, the name of the objects concerned in the source schema and the name of the new objects in the target schema. More precisely, the history of a compound transformation contains the signatures of each transformation according to their order in the chain. For example, the signature of the transformations represented in Figure 5 is:

`T:(EA2,R)⟵ATTRIBUTE-to-ET/instance(A,A2)`
`Ti:(A2)⟵ET-to-ATTRIBUTE(EA2)`

The first expression can be read as follows: by application of the `ATTRIBUTE-to-ET/instance` transformation on attribute `A2` of entity type `A`, a new entity type `EA2` and a new relationship type `R` are created. To simplify, certain objects implied in the transformation are not specified in the signature. This is the case for the relationship type `R` which disappears when one applies `ET-to-ATTRIBUTE`.

3.3.4. Source and target physical mappings

The mappings between the source and target physical schemas are modelled through a transformation history. Two transformation histories can be defined according to the nature of schema transformation (physical or conceptual). The history for the physical schema strategy is modelled by a compound transformation: **`SPS-to-TPS`** in such a way that `TPS = SPS-to-TPS(SPS)`. On the other hand, the history for the conceptual schema strategy is defined by the complex compound transformation: **`SPS-to-CS-to-TPS = SPS-to-CS o CS-to-TPS`** in such a way that `TPS = CS-to-TPS(SPS-to-CS(SPS))`.

4. Data conversion

4.1. Principle

Schema conversion is concerned with the conversion of data format and not of its content [1]. Data conversion is taken in charge by a software called the *converter*, or *Extract-Transform-Load* processor (see Figure 6), which transforms the data from the data source to the format defined by the target schema. A converter has three main

functions. Firstly, it performs the extraction of the source data. Then, it converts these data in such a way that their structures match the target (new) format. Finally, it writes legacy data in the target format.

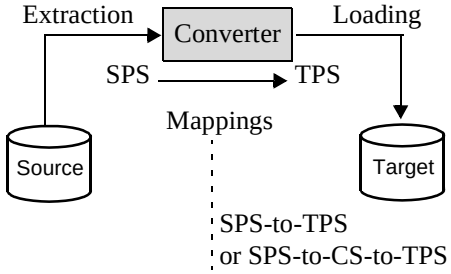


Figure 6. Data migration architecture: converter and schema transformation.

A converter relies on the mappings between the source and target physical schemas. More precisely, this mapping is made up of the instance mappings (`t`) of the source-to-target transformations stored in the history.

The conceptual schema conversion strategy (D2) recovers the conceptual schema (CS) and the target physical schema (TPS) implements all the constraints of this schema. It is important to notice that the legacy data often violate some constraints of CS discovered during the schema refinement. Thus the data migration is impossible if the legacy data are not conformed to the CS constraints. So the erroneous data must be corrected before the data migration process. This correction step is a tedious task that is generally not automatized. The systematic approach of the physical schema conversion strategy (D1) does not require this data cleaning step. Indeed, the only known constraints are recovered by the DDL analysis and they are verified by the legacy DMS. In this context, TPS only contains the same constraints thus the erroneous data can be propagated in the target database.

4.2. Methodology

Data conversion involves three main tasks. Firstly, the target physical schema (`TPS`) must be implemented in the new DMS. Secondly, the mapping between the source and target physical schemas must be defined as sequences of schema transformations according to one of the two strategies described in Section 3. Finally, these mappings must be implemented in the converter for translating the legacy data according to the format defined in `TPS`.

Since each transformation is formally defined (<T,t>), the instance mapping `sps-to-tps` is automatically derived from the chain `SPS-to-TPS` forming the history of the process that builds TPS from SPS. The converter is based on the structural mappings (`SPS-to-TPS` or `SPS-to-CS-to-TPS`) to write the extraction and insertion requests, and on the instance mappings (`sps-to-tps` or `sps-to-cs-to-tps`) for the data conversion.

4.3. Support

Writing data converters manually is an expensive task, particularly for complex mappings (for simple mappings parametric converters are quite sufficient). The DB-MAIN CASE tool includes specific history analyzers and converter generators that have been described in [5].

5. Program conversion

When the structure of the database changes, the application programs and more specifically the access statements must be changed accordingly. This section analyzes the three program modification strategies specified in Figure 1. Each strategy is presented in the framework of a schema conversion strategy. The first one relies on the *Conceptual schema conversion strategy (D2)* and on *wrapper* technology (P1) to map the access primitives onto the new database (DMS-DML statements are replaced with calls to the wrapper). The second strategy (P2) relies on the *Physical schema conversion strategy* (D1) that allows merely replacing each statement with its equivalent in the new DMS-DML. According to the third strategy <D2,P3>, the access logic is rewritten to comply to the new DMS-DML. In the last two strategies, the access statements are expressed into the DML of the new DMS.

5.1. Wrapper strategy <D2,P1>

5.1.1. Principle

The wrapper migration attempts to keep the legacy programs logic and to map it on the new DMS technology [4]. A *data wrapper* is a *data model conversion* component that is called by the application program to carry out operations on the database. Its goal is generally to provide application programs with a modern interface to a legacy database (e.g., allowing Java programs to access COBOL files). In the present context, the wrapper is to transform the renovated data into the legacy format, e.g., to allow COBOL programs to *read*, *write*, *rewrite* records that are built from rows extracted from a relational database.

In this way, the application program invokes the wrapper instead of the legacy DMS. If the wrapper simulates the modeling paradigm of the legacy DB and its interface, the alteration of the legacy code is minimal. It consists in replacing the DML statements with wrapper invocations.

The wrapper converts all legacy DMS requests from legacy applications into requests against the new DMS that now manages the data. Conversely, it captures results from

the new DMS, possibly converts them to the appropriate legacy format [13] (Figure 7) and delivers them to the application program.

Some legacy DMS, such as MicroFocus COBOL, provide an elegant way to interface wrappers with the legacy code. They allow programmers to replace the standard DMS library with customized library. In this case, the legacy code needs not be altered at all.

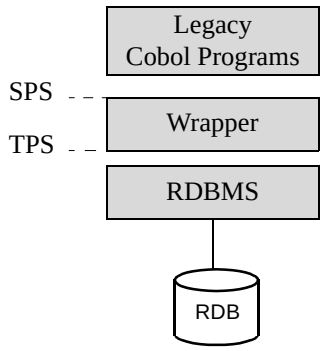


Figure 7. Wrapper-based migration architecture: a wrapper allows the data managed by a new DMS to be accessible by the legacy programs.

5.1.2. Methodology

As part of the InterDB project, we have developed a technology for the automated production of hard-coded wrappers for data systems. In [15], we have shown that the code of a wrapper dedicated to a couple of DMS families, performs the structural and instance mappings and that these mappings can be modelled through semantics-preserving transformations. For a consultation query, the structural mapping explains how to translate the query while the instance mapping explains how to form the result instances. Consequently the `SPS-to-TPS` (or `SPS-to-CS-to-TPS`) mapping and its instance counterpart `sps-to-tps` (`sps-to-cs-to-tps`) provide the necessary and sufficient information to produce the procedural code of a specific wrapper and to build a common generator of wrappers for a specific couple of source and target DMS.

5.1.3. Support

As explained above, a wrapper generator needs the description of the source and target physical schemas as well as the history of their derivation. To simplify, the wrapper generation in the DB-MAIN environment is performed in two steps, namely the history analysis and the wrapper encoding. The history analyzer parses the schema transformation history in order to yield functional specifications from which the target/source physical schemas are enriched with source/target physical correspondences. The end product of this phase is target/source physical schemas that include, for each construct, the way it has been mapped onto the other physical schema constructs. In this way, each schema holds all the information required by the generator. From the TPS (or the SPS), the wrapper encoder produces the procedural code of the specific wrapper and a documentation for the programmers.

5.1.4. Illustration

To illustrate the way data wrappers are used, let us consider the following statements of the legacy Cobol program:

```
READ PRODUCT KEY IS PROD-CODE
  INVALID KEY GO TO ERR-123.
DELETE PRODUCT END-DELETE.
```

Each primitive is replaced by the calling of the wrapper to ask it to carry out a similar function. This substitution yields the following piece of code:

```
CALL WR-ORD-MNGMT
  USING "READKEY","PRODUCT","PROD-CODE",
    PRODUCT,WR-STATE.
IF STATUS OF WR-STATE NOT = 0 THEN
  GO TO ERR-123.
CALL WR-ORD-MNGMT
  USING "DELETE","PRODUCT","",
    PRODUCT,WR-STATE.
```

Though describing the wrapper interface in detail is beyond the scope of this paper, we will only mention that the wrapper is stateless, in that it keeps no information on the application program activities. The state of the program is stored in the state block `WR-STATE` which records, among others, the current key, the current record ID and the status of the last primitive (`STATUS`). The wrapper executes each legacy primitive as a sequence of SQL operations that retrieve the rows forming the data record or that delete the components of this record. This translation is based on the transformation history and the source and target physical schemas. For instance, in the *Conceptual schema conversion* strategy, the wrapper builds the ORDER record by joining the identified `ORDER` table row with the dependent `DETAIL` rows (Figure 4).

5.2. Statement rewriting strategy <D1,P2>

5.2.1. Principle

This program modification technique depends on the Physical schema conversion strategy. It consists in replacing the legacy DMS-DML statements with DML statements of the new DMS. For example, every file access in a COBOL program must be replaced by a relational `SFW` statement. Due to the simple `SPS-TPS` mapping this process can be largely automated.

5.2.2. Methodology

The program modification process is fairly simple and needs no elaborated methodology. Each DML statement

must be located, its parameters must be identified and the new DML statement sequence must be defined and inserted in the code. The main point is how to translate iterative accesses in a systematic way. For instance, in the most popular *COBOL to SQL* conversion, there are several techniques to express the typical *start/read-next* loop with SFW statements. The task may be complex due to loosely structured programs and the use of dynamic DML statements. For instance, a COBOL READ NEXT statement can follow a statically unidentified START initial statement, making impossible the identification of the record key used. A description of a specific technique that solves this problem will be found in section 5.2.4.

5.2.3. Support

This technique requires program analyzers and text substitution engines. Several program migration *factories* are proposed by software companies. Most of them comply with the Statement rewriting strategy.

5.2.4. Illustration

The change of paradigm when moving from standard files to relational database induces problems such as the identification of the sequence scan. COBOL allows the programmer to start a sequence based on an indexed key (START-KEY and READ-KEY), then to go on in this sequence through READ-NEXT primitives. The most obvious SQL translation is through a cursor-based loop. However, the READ-NEXT statements can be scattered throughout the program, so that the identification of the initiating START statement is only possible through complex static analysis of the program data and control flows.

The technique illustrated below is based on state registers, such as ORD-SEQ, that specify the current key of each record type, and consequently the matching SQL cursor. A cursor is declared for each kind of record key usage (EQUAL, GREATER, NOT LESS) in the program. For instance, the table ORD gives at most six cursors (combination of two record keys and three key usages).

The COBOL access loop

```
  MOVE CUS-CODE TO ORD-CUSTOMER.
  START ORDER KEY >= ORD-CUSTOMER.                x
  MOVE 0 TO END-FILE.
  PERFORM READ-ORD UNTIL END-FILE = 1.
READ-ORD SECTION.
BEG-ORD.
  READ ORDER NEXT                                 y
    AT END MOVE 1 TO END-FILE GO TO EXIT-ORD.     z
  <<processing current ORD record>>
EXIT-ORD.
  EXIT.
```

The SQL access loop

```
  EXEC SQL declare cursor ORD_GE_K1 for
  select ORD_CODE,ORD_CUSTOMER,ORD_DETAIL
  from ORDER where ORD_CODE >= :ORD-CODE
  order by ORD_CODE END-EXEC.
  . . .
  EXEC SQL declare cursor ORD_GE_K2 for           x
  select ORD_CODE,ORD_CUSTOMER,ORD_DETAIL         x
  from ORDER where ORD_CUSTOMER >= :ORD-CUSTOMER  x
  ORDER BY ORD_CUSTOMER END-EXEC.                 x
  ...
  MOVE CUS-CODE TO ORD-CUSTOMER.
  EXEC SQL open ORD_GE_K2 END-EXEC.               x
  MOVE "ORD_GE_K2" to ORD-SEQ.                    x
  MOVE 0 TO END-FILE.
  PERFORM READ-ORD UNTIL END-FILE = 1.
READ-ORD SECTION.
BEG-ORD.
  IF ORD-SEQ = "ORD_GE_K1"                        y
  THEN                                            y
  EXEC SQL fetch ORD_GE_K1 into :ORD-CODE,        y
       :ORD-CUSTOMER,:ORD-DETAIL END-EXEC         y
  ELSE IF ORD-SEQ = "ORD_GE_K2"                   y
  THEN                                            y
  EXEC SQL fetch ORD_GE_K2 into :ORD-CODE,        y
       :ORD-CUSTOMER,:ORD-DETAIL END-EXEC         y
  ELSE IF ...                                     y
  END-IF.                                         y
  IF SQLCODE NOT = 0 THEN                         z
     MOVE 1 TO END-FILE GO TO NO-ORD.             z
  <<processing current ORD record>>
EXIT-ORD.
  EXIT.
```

5.3. Logic rewriting strategy <D2,P3>

5.3.1. Principle

The program is rewritten to explicitly access the new data structures and take advantage of the new data system features. This rewriting task is a complex conversion process that requires an in-depth understanding of program logic. For example, the processing code of a COBOL record type may be replaced by a code section that copes with several SQL tables or a COBOL loop may be replaced with a single SQL join.

The complexity of the problem prevents the complete automation of the conversion process. Tools can be developed to find the statements to be modified by the programmer and to give hints on how to rewrite them. However, actually modifying the code is up to the programmer.

5.3.2. Methodology

This strategy is much more complex than the previous one since every part of the program may be influenced by the schema transformation.

The most obvious method consists in (1) identifying the file access statements, (2) identifying the statements and the data objects that depend on these access statements and (3) rewriting these statements and redefining these data objects.

5.3.3. Support

The identification tasks 1 and 2 relate to the program understanding realm, where such techniques as searching for *clichés*, variable dependency analysis and program slicing ([11], [17] for information) often are favorite weapons. The supporting tools must include engines for these activities.

5.3.4. Illustration

We illustrate this strategy by rewriting two code sections from the COBOL program mentioned so far. The first example displays information on an order together with some data from its customer. In the SQL program, this pattern clearly suggests joining the tables `ORDER` and `CUSTOMER` then extracting the desired column values.

The source COBOL section

```
DISP-ORD.
  READ ORDER KEY IS ORD-CODE
  INVALID KEY GO TO ERR-CUS-NOT-FOUND.
  PERFORM DISP-ORD-CUS-NAME.
DISP-ORD-CUS-NAME.
  MOVE ORD-CUSTOMER TO CUS-CODE
  READ CUSTOMER INVALID KEY
    DISPLAY "ERROR : UNKOWN CUSTOMER"
  NOT INVALID KEY
    DISPLAY "ORD-CODE: " ORD-CODE NAME.
```

The target SQL section

```
DISP-ORD.
  EXEC SQL
    SELECT O.CODE, C.NAME INTO :ORD-CODE, :NAME
    FROM ORDER O, CUSTOMER C
    WHERE O.CUS_CODE = C.CODE
        AND O.CODE = :ORD-CODE
  END-EXEC.
  IF SQLCODE = 0 THEN
    DISPLAY "ORD-CODE: " ORD-CODE NAME
  ELSE
    GO TO ERR-CUS-NOT-FOUND.
```

The second example has been used in the section dealing with the *Wrapper* strategy. The two-step *position then delete* pattern, which is typical of navigational DMS, can be replaced with a single set expression of SQL.

The source COBOL section

```
   READ PRODUCT KEY IS PROD-CODE
     INVALID KEY GO TO ERR-123.
   DELETE PRODUCT END-DELETE.
```

The target SQL statement

```
   EXEC SQL
    DELETE FROM PRODUCT WHERE CODE = :PROD-CODE
   END-EXEC.
   IF SQLCODE NOT = 0 THEN GO TO ERR-DEL-PROD.
```

6. Conclusion

The variety in corporate requirements as far as system reenginering is concerned naturally leads to an equally large spectrum of migration strategies. This paper has identified two main independent lines of decision, the first one about database conversion (schema and contents) and the second one on program conversion. From them, it has identified and analyzed in details three representative strategies.

Though comparing them would not be easy, we will try to give some hints on their respective advantages and drawbacks. Clearly, as far as quality is required, the best strategy in database conversion is the *Conceptual schema conversion* (D2), since it yields a high quality normalized database that is no longer impaired by historical and obsolete decisions, and by the legacy of inadequate technologies. The data structure conversion based on the complete reverse engineering process is complex but uses all the expressiveness of the target DMS. Future modifications on the new system are easier because the new data system does not contain the physical artifacts of the legacy system and its semantics are well known.

Clearly too, the *Physical schema conversion* (D1) is the worst decision. It does not recover the source physical schema semantics and blindly translates the technical structures peculiar to source technology or design errors in the target structures. However, since it is by far less expensive, it is also the most popular. It does not require to carry out a complete DBRE process. Only an automatic physical schema recovering is required. The evolution of the target structures is often complex or impossible because program modifications require knowledges of source data system semantics and physical artifacts.

Program conversion is a less mature domain, particularly when it derives from database changes. Though many technical and architectural approaches exist, we have found it interesting to describe three representative strategies through which we can identify some of the most challenging problems.

In terms of quality, the *Statement rewriting* strategy (P2) seems to degrade the program readability but is fairly inexpensive, while the *Logic rewriting* strategy (P3) modernizes the programs (their logic matches the paradigm of the new DMS) by keeping them as clear and intuitive as their source version. The *Wrapper* strategy (P1) allows the program to be kept, with their origin qualities and problems, while working on the new database.

In terms of programming efforts, the *Statement rewriting* strategy is achieved by an automatic processor and the wrappers of the strategy P1 are automatically generated from the mapping between source and target physical structures. These strategies can be a solution within the framework of an incremental migration. Unlike the two others, significant programmer interventions as well as the use of complex program comprehension tools are necessary in the *Logic rewriting* strategy. The programming

effort is only interesting if the migration is limited to the data system.

7. References

[1] Aiken, P., Muntz, A., Richards, R.: "DOD Legacy Systems - Reverse-Engineering Data Requirements", *Communications of the ACM*, May 1994.

[2] Behm, A., Geppert, A., Dittrich, K.R.: "On the migration of Relational Schemas and Data to Object-Oriented Database Systems", in *Proceedings of Re-Technologies in Information Systems*, Klagenfurt, Austria, December 1997.

[3] Bisbal, J., Lawless, D. , Wu, B., Grimson, J.: "Legacy Information Systems: Issues and Directions", *IEEE Software*, September/October 1999.

[4] Brodie, M. L., Stonebraker, M.: *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*, Morgan Kaufmann, 1995.

[5] Delcroix, C., Thiran, Ph., Hainaut, J.-L.: "Approche Transformationnelle de la Ré-ingénierie des Données", *Ingénierie des Systèmes d'Information*, Hermes-Sciences, Paris, December 2001.

[6] Jahnke, J.-H., Wadsack, J. P.: "Varlet: Human-Centered Tool Support for Database Reengineering", in *Proceedings of Workshop on Software-Reengineering (WCRE'99)*, May 1999.

[7] Jeusfeld, M. A., Johnen, U. A.: "An Executable Meta Model for Re-Engineering of Database Schemas", in *Proceedings of Conference on the Entity-Relationship Approach*, Manchester, December 1994.

[8] Hainaut, J.-L., Roland, D., Hick, J.-M., Henrard, J., Englebert, V.: "Database Reverse Engineering: from Requirements to CARE Tools", *Journal of Automated Software Engineering*, 3(1), 1996.

[9] Hainaut, J.-L.: "Specification preservation in schema transformations - Application to semantics and statistics", *Data & Knowledge Engineering*, 16(1), Elsevier Science Publish, 1996.

[10] Hainaut, J.-L., Roland, D., Englebert, V., Hick, J.-M., Henrard, J.: "Database Reverse Engineering - A Case Study", *WCRE'97*, Amsterdam, 1997.

[11] Henrard, J., Englebert, V., Hick, J.-M., Roland, D., Hainaut, J.-L.: "Program understanding in databases reverse engineering", *DEXA'98*, Vienna, 1998.

[12] Menhoudj, K., Ou-Halima, M.: "Migrating Data-Oriented Applications to a Relational Database Management System", in *Proceedings of the Third International Workshop on Advances in Databases and Information Systems*, Moscow, 1996.

[13] Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J.: "A Query Translation Scheme for Rapid Implementation of Wrappers", in *Procedings of the International Conference on Deductive and Object-oriented Databases*, 1995.

[14] Renaissance (ESPRIT project): *Client/Server Migration: Guidelines for Migration from Centralised to Distributed Systems*, consultancy report, 1998.

[15] Thiran, Ph., Hainaut, J.-L.: "Wrapper Development for Legacy Data Reuse", in *Proceedings of WCRE'01*, IEEE Computer Society Press, 2001.

[16] Tilley, S. R., Smith, D. B.: "Perspectives on Legacy System Reengineering", technical report, Software Engineering Institute, Carnegie Mellon University, 1995.

[17] Weiser, M.: "Program Slicing", *IEEE TSE*, 10, pp. 352-357, 1984.

[18] Wiederhold, G.: "Modeling and System Maintenance", in *Proceedings of the International Conference on Object-Oriented and Entity-Relationship Modeling*, Berlin, 1995.

[19] Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., O'Sullivan, D.: "The Butterfly Methodology: A Gateway-free Approach for Migrating Legacy Information Systems", in *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems*, Italy, September 1997.

Strategy	Data engineering	Program engineering	Quality	Performance	Maintenance	Evolutivity
D1, P2	cheap, fully automated	cheap, fully automated	poor DB and the programs are not changed	poor, the legacy structures are simulate into the new DB	difficult, the semantics of the DB is not recovered	difficult, the DB simulate the legacy one
D2, P1	expensive, require a complete DBRE	cheap, fully automated	DB of good quality, the programs are not changed	poor, the legacy data access are simulated into the new DB	like the legacy system, but the semantics of the DB is known	easier, the new functions can directly access to the new DB
D2, P3		very expensive, require a deep understanding of the programs	DB of good quality, the programs have been rewritten	good use of the expressiveness of the new DBMS	easier, the semantics of the DB is known	

Figure 8. A comparison of the different strategy.