6

Migration of Legacy Information Systems

Jean-Luc Hainaut[1], Anthony Cleve[1], Jean Henrard[2], and Jean-Marc Hick[2]

[1] PReCISE Research Centre, University of Namur, Belgium
[2] REVER s.a., Charleroi, Belgium

Summary. This chapter addresses the problem of platform migration of large business applications, that is, complex software systems built around a database and comprising thousands of programs. More specifically, it studies the substitution of a modern data management technology for a legacy one. Platform migration raises two major issues. The first one is the conversion of the database to a new data management paradigm. Recent results have shown that automated lossless database migration can be achieved, both at the schema and data levels. The second problem concerns the adaptation of the application programs to the migrated database schema and to the target data management system. This chapter first poses the problem and describes the State of the Art in information system migration. Then, it develops a two-dimensional reference framework that identifies six representative migration strategies. The latter are further analysed in order to identify methodological requirements. In particular, it appears that transformational techniques are particularly suited to drive the whole migration process. We describe the database migration process, which is a variant of database reengineering. Then, the problem of program conversion is studied. Some migration strategies appear to minimise the program understanding effort, and therefore are sound candidates to develop practical methodologies. Finally, the chapter describes a tool that supports such methodologies and discusses some real-size case studies.

6.1 Introduction

Business applications are designed as informational and behavioural models of an organization, such as an enterprise or an administration, and are developed to efficiently support its main business processes. The term *information system* is often used to designate large-scale business applications. Though this term has been given several interpretations, we will limit its scope in this chapter to a complex software and information system comprising one or several databases and programs, whatever their technologies, that support the organization's business processes. In particular, we will ignore other important components such as user interfaces as well as distribution and cooperation frameworks. The information system relies on a technological platform, made up of such components as operating systems, programming languages and database management systems.

T. Mens, S. Demeyer (eds.), *Software Evolution*.
DOI 10.1007/978-3-540-76440-3, © Springer 2008

6.1.1 Information System Evolution

Since every organization naturally evolves over time, its information system has to change accordingly. This evolution is often driven by the business environment, that forces the organization to change its business processes, and, transitively, the information system that supports them. Practically, the integration of new concepts and new business rules generally translates into the introduction of new data structures and new program components, or into the updating of existing ones.

On the other hand, the rapid technological evolution also induces a strong pressure to modify the information system in order to make it apt to support new requirements that the legacy technological platform was unable to meet. Two common motivations are worth being mentioned, namely flexibility and vanishing skills.

Flexibility. As summarised by Brodie and Stonebraker [84], *a legacy Information System is any Information System that significantly resists modifications and change*. One of the most challenging instances of this problem comes from the increasing requirement to answer, almost in real time, unplanned questions by extracting data from the database. COBOL file managers, as well as most legacy data management technologies, are efficient for batch and (to some extent) transaction processing. However, answering a new query requires either extending an existing program or writing a new one, an expensive task that may need several days. On the contrary, such a query can be formulated in SQL on a relational database in minutes, most often by non-expert users.

Skill shortage. Many core technologies enjoy a surprisingly long life, often encompassing several decades. Hiring experts that master them has become more and more difficult, so that companies may be forced to abandon otherwise satisfying technologies due to lack of available skills.

The business and technological dimensions of evolution can be, to a large extent, studied independently. In this chapter, we address the issue of adapting an information system to technological changes, a process generally called *migration*. More precisely we will study the substitution of a modern data management system for a legacy technology.

6.1.2 Information System Reengineering and Migration

As defined by Chikofsky and Cross [112], *reengineering, also known as [...] renovation [...], is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some more form of forward engineering or restructuring*. Migration is a variant of reengineering in which the transformation is driven by a major technology change.

Replacing a DBMS with another one should, in an ideal world, only impact the database component of the information system. Unfortunately, the database most often has a deep influence on other components, such as the application programs.

Two reasons can be identified. First, the programs invoke data management services through an API that generally relies on complex and highly specific protocols. Changing the DBMS, and therefore its protocols, involves the rewriting of the invocation code sections. Second, the database schema is the technical translation of its conceptual schema through a set of rules that is dependent on the DBMS data model. Porting the database to another DBMS, and therefore to another data model, generally requires another set of rules, that produces a significantly different database schema. Consequently, the code of the programs often has to be adapted to this new schema. Clearly, the renovation of an information system by replacing an obsolete DBMS with a modern data management system leads to non trivial database (schemas and data) and programs modifications.

6.1.3 System Migration: State of the Art

Technically, a legacy information system is made up of large and ageing programs relying on legacy database systems (like IMS or CODASYL) or using primitive DMSs[3] (a.o., COBOL file system, ISAM). Legacy information systems often are isolated in that they do not easily interface with other applications. Moreover, they have proved critical to the business of organizations. To keep being competitive, organizations must improve their information system and invest in advanced technologies, specially through system evolution. In this context, the claimed 75% cost of legacy systems maintenance (w.r.t. total cost) is considered prohibitive [541].

Migration is an expensive and complex process, but it greatly increases the information system control and evolution to meet future business requirements. The scientific and technical literature ([69, 84]) mainly identifies two migration strategies, namely rewriting the legacy information system from scratch or migrating by small incremental steps. The incremental strategy allows the migration projects to be more controllable and predictable in terms of calendar and budget. The difficulty lies in the determination of the migration steps.

Legacy IS migration is a major research domain that has yielded some general migration methods. For example, Tilley and Smith [500] discuss current issues and trends in legacy system reengineering from several perspectives (engineering, system, software, managerial, evolutionary, and maintenance). They propose a framework to place reengineering in the context of evolutionary systems. The butterfly methodology proposed by Wu et al. [546] provides a migration methodology and a generic toolkit to aid engineers in the process of migrating legacy systems. This methodology, that does not rely on an incremental strategy, eliminates the need of interoperability between the legacy and target systems.

Below, we gather the major migration approaches proposed in the literature according to the various dimensions of the migration process as a whole.

[3] DMS: Data Management System.

Language Dimension

Language conversion consists in translating (parts of) an existing program from a source programming language to a target programming language. Ideally, the target program should show the same behaviour as the source program. Malton [342] identifies three kinds of language conversion scenarios, with their own difficulties and risks:

- **Dialect conversion** is the conversion of a program written in one dialect of a programming language to another dialect of the same programming language.
- **API migration** is the adaptation of a program due to the replacement of external APIs. In particular, API migration is required when changing the data management system.
- **Language migration** is the conversion from one programming language to a different one. It may include dialect conversion and API migration.

Two main language conversion approaches can be found in the literature. The first one [535], that might be called *abstraction-reimplementation*, is a two-step method. First, the source program is analysed in order to produce a high-level, language-independent description. Second, the reimplementation process transforms the abstract description obtained in the first step into a program in the target language. The second conversion approach [493, 342] does not include any abstraction step. It is a three-phase conversion process: (1) *normalization*, that prepares the source program to make the translation step easier; (2) *translation*, that produces an equivalent program that correctly runs in the target language; (3) *optimization*: that improves the maintainability of the target source code.

Terekhov and Verhoef [493] show that the language conversion process is far from trivial. This is especially true when the source and the target languages come from different paradigms. A lot of research has been carried out on specific cases of language conversion, among which PL/I to C++ [290], Smalltalk to C [558], C to Java [350] and Java to C# [158].

User Interface Dimension

Migrating user interfaces to modern platforms is another popular migration scenario. Such a process may often benefit from an initial reverse engineering phase, as the one suggested by Stroulia et al. [478]. This method starts from a recorded trace of the user interaction with the legacy interface, and produces a corresponding state-transition model. The states represent the unique legacy interface screens while the transitions correspond to the user action sequences enabling transitions from one screen to another. De Lucia et al. [333] propose a practical approach to migrating legacy systems to multi-tier, web-based architectures. They present an Eclipse-based plugin to support the migration of the graphical user interface and the restructuring and wrapping of the original legacy code.

Platform and Architecture Dimensions

Other researches, that we briefly discuss below, examine the problem of migrating legacy systems towards new architectural and technological platforms.

Towards distributed architectures. The Renaissance project [534] develops a systematic method for system evolution and re-engineering and provides technical guidelines for the migration of legacy systems (e.g., COBOL) to distributed client/server architectures. A generic approach to reengineering legacy code for distributed environments is presented by Serrano et al. [458]. The methodology combines techniques such as data mining, metrics, clustering, object identification and wrapping. Canfora et al. [101] propose a framework supporting the development of thin-client applications for limited mobile devices. This framework allows Java AWT applications to be executed on a server while the graphical interfaces are displayed on a remote client.

Towards object-oriented platforms. Migrating legacy systems towards object-oriented structures is another research domain that has led to a lot of mature results, especially on object identification approaches ([560, 99, 517, 201, 449]). Regarding the migration process itself, the approach suggested by De Lucia et al. [144] consists of several steps combining reverse engineering and reengineering techniques. More recently, Zou and Kontogiannis [569] have presented an incremental and iterative migration framework for reengineering legacy procedural source code into an object-oriented system.

Towards aspect-orientation. System migration towards aspect-oriented programming (AOP) still is at its infancy. Several authors have addressed the initial reverse engineering phase of the process, called *aspect mining*, which aims at identifying crosscutting concern code in existing systems. Among the various aspect mining techniques that have been proposed, we mention *fan-in analysis* [348], *formal concept analysis* [506], *dynamic analysis* [503] and *clone detection* [93]. Regarding clone detection, Chapter 2 provides an overview of techniques to identify and remove software redundancies. We also refer to Chapter 9 for a more complete discussion about current issues as well as future challenges in the area of aspect mining, extraction and evolution.

Towards service-oriented architectures. Migrating legacy systems towards service-oriented architectures (SOA) appears as one of the next challenges of the maintenance community. Sneed [465] presents a wrapping-based approach according to which legacy program functions are offered as web services to external users. O'Brien et al. [400] propose the use of architecture reconstruction to support migration to SOA. Chapter 7 presents a tool-supported methodology for migrating legacy systems towards three-tier and service-oriented architectures. This approach is based on graph transformation technology.

Database Dimension

Closer to our data-centred approach, the Varlet project [249] adopts a typical two phase reengineering process comprising a reverse engineering process phase followed by a standard database implementation. The approach of Jeusfeld [256] 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 [361] present a method to migrate the data of COBOL legacy system into a relational database management system. The hierarchical to relational database migration is discussed in [360, 359]. General approaches to migrate relational database to object-oriented technology are proposed by Behm et al. [53] and Missaoui et al. [373]. More recently, Bianchi et al. [62] propose an iterative approach to database reengineering. This approach aims at eliminating the ageing symptoms of the legacy database [527] when incrementally migrating the latter towards a modern platform.

Related Work Limitations

Though the current literature on data-intensive systems migration sometimes recommend a semantics-based approach, relying on reverse engineering techniques, most technical solutions adopted in the industry are based on the so-called *one-to-one* migration of the data structures and contents, through a fully-automated process. As we will see below, these approaches lead to poor quality results. Secondly, while most papers provide ad hoc solutions for particular combinations of source/target DB platforms, there is still a lack of generic and systematic studies encompassing database migration strategies and techniques. Thirdly, the conversion of application programs in the context of database migration still remains an open problem. Although some work (e.g., [62]) suggests the use of wrapping techniques, very little attention is devoted to the way database wrappers are built or generated. In addition, the impact of the different conversion techniques on target source code maintainability has not been discussed.

6.1.4 About This Chapter

This chapter presents a practical approach to data-intensive application reengineering based on two independent dimensions, namely the data and the programs. We first propose a reference model that allows us to describe and compare the main migration approaches that are based on DBMS substitution (Section 6.2). This model identifies six representative strategies [228]. Section 6.3 develops a transformational framework that forms a sound basis to formalise database and program evolution, including migration. Then, the conversion of three main components of the information system, namely database schemas, database contents and programs, are described and discussed in Sections 6.4, 6.5 and 6.6 respectively. Section 6.7 describes a prototype CASE environment for information system migration while Section 6.8 discusses some experimental results. The six reference migration strategies are compared in Section 6.9. Finally, Section 6.10 draws some conclusions and suggests paths for future work.

To make the discussion more concrete, we base it on one of the most popular problem patterns, that is, the conversion of a legacy COBOL program, using standard

indexed files, into an equivalent COBOL program working on a relational database. The principles of the discussion are of course independent of the language and of the DMS.

6.2 Migration Reference Model

There is more than one way to migrate a data-intensive software system. Some approaches are quite straightforward and inexpensive, but lead to poorly structured results that are difficult to maintain. Others, on the contrary, produce good quality data structures and code, but at the expense of substantial intelligent (and therefore difficult to automate) code restructuring. We have built a reference model based on two dimensions, namely data and programs. Each of them defines a series of change strategies, ranging from the simplest to the most sophisticated. This model outlines a solution space in which we identify six typical strategies that will be described below and discussed in the remainder of the chapter. This model relies on a frequently used scenario, called *database-first* [545], according to which the database is transformed before program conversion. This approach allows developers to cleanly build new applications on the new database while incrementally migrating the legacy programs.

Information system migration consists in deriving a new database from a legacy database and in further adapting the software components accordingly [84]. Considering that a database is made up of two main components, namely its schema(s) and its contents (the *data*), the migration comprises three main steps: (1) *schema conversion*, (2) *data conversion* and (3) *program conversion*. Figure 6.1 depicts the organization of the database-first migration process, that is made up of subprocesses that implement these three steps. Schema conversion produces a formal description of the mapping between the objects of the legacy (S) and renovated (S') schemas. This mapping is then used to convert the data and the programs. Practical methodologies differ in the extent to which these processes are automated.

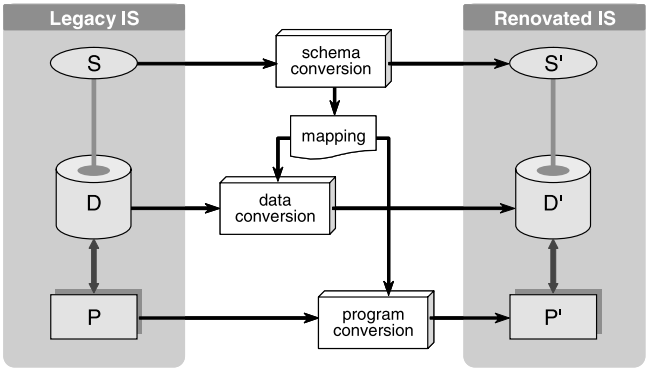

Fig. 6.1. Overall view of the *database-first* information system migration process

- **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. Most generally, the conversion of a source schema into a target schema is made up of two processes. The first one, called database reverse engineering [215], aims at recovering 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. Each of these processes can be modelled by a chain of semantics-preserving schema transformations.
- **Data conversion** is the migration of the data instance from the legacy database to the new one. This migration involves data transformations that derive from the schema transformations described above.
- **Program conversion**, in the context of database migration, 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.

6.2.1 Strategies

We consider two dimensions, namely database conversion and program conversion, from which we will derive migration strategies.

The Database dimension (D)

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 expressiveness of the new DMS model and is fully documented, but, as expected, the process is more expensive.

The program dimension (P)

Once the database has been converted, several approaches to application programs adaptation can be followed. We identify three reference 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 or CODASYL/IMS

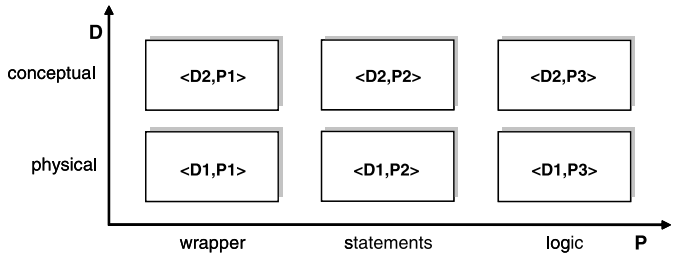

Fig. 6.2. The six reference IS migration strategies

databases, generally through program calls instead of through native I/O 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[4]. 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 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 due to, for instance, the change in database paradigm. These dimensions define six reference information system migration strategies (Figure 6.2).

6.2.2 Running Example

The strategies developed in this chapter will be illustrated by a small case study in which the legacy system comprises a standalone COBOL program and three files. Despite its small size, the files and the program exhibit representative instances of the most problematic patterns. 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.

6.3 The Transformational Approach

Any process that consists in deriving artefacts from other artefacts relies on such techniques as renaming, translating, restructuring, replacing, refining and abstracting, which basically are *transformations*. Most database engineering processes can be formalised as chains of elementary schema and data transformations that preserve some of their aspects, such as its information contents [217]. Information system evolution, and more particularly system migration as defined in this chapter, consists

[4] DML: Data Manipulation Language.

of the transformation of the legacy database and of its programs into a new system comprising the renovated database and the renovated programs. As far as programs are concerned, the transformations must preserve the behaviour of the interface with the database management system, though the syntax of this interface may undergo some changes. Due to the specific scope of the concept of migration developed here, only simple program transformations will be needed.

6.3.1 Schema Transformation

Roughly speaking, an elementary 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 by an equivalent entity type or by a foreign key and replacing an attribute by an entity type (Figure 6.3) are some examples of schema transformations.

More formally, a transformation Σ is defined as a couple of mappings $<T,t>$ such that, $C' = T(C)$ and $c' = t(c)$, where c is any instance of C and c' the corresponding instance of C'. *Structural mapping* T is a rewriting rule that specifies how to modify the schema while *instance mapping* t states how to compute the instance set of C' from the instances of C.

There are several ways to express mapping T. For example, T can be defined (1) as a couple of predicates defining the minimal source precondition and the maximal target postcondition, (2) as a couple of source and target patterns or (3) through a procedure made up of removing, adding, and renaming operators acting on elementary schema objects. Mapping t will be specified by an algebraic formula, a calculus expression or even through an explicit procedure.

Any transformation Σ can be given an inverse transformation $\Sigma' = <T',t'>$ such that $T'(T(C)) = C$. If, in addition, we also have: $t'(t(c)) = c$, then Σ (and Σ') are called semantics-preserving[5]. Figure 6.3 shows a popular way to convert an attribute into an entity type (structural mapping T), and back (structural mapping T'). The instance mapping, that is not shown, would describe how each instance of source attribute A2 is converted into an EA2 entity and an R relationship.

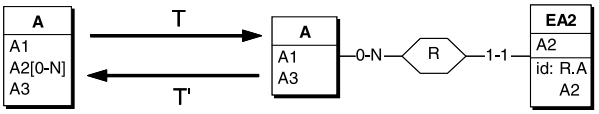

Fig. 6.3. Pattern-based representation of the structural mapping of ATTRIBUTE-to-ET transformation that replaces a multivalued attribute (A2) by an entity type (EA2) and a relationship type (R)

[5] The concept of semantics (or information contents) preservation is more complex, but this definition is sufficient in this context. A more comprehensive definition can be found in [217].

Practically, the application of a transformation will be specified by its signature, that identifies the source objects and provides the names of the new target objects. For example, the signatures of the transformations of Figure 6.3 are:

T : (EA2,R) ← ATTRIBUTE-to-ET(A,A2)
T' : (A2) ← ET-to-ATTRIBUTE(EA2)

Transformations such as those in Figure 6.3 include names (A, A1, R, EA2, etc.) that actually are variable names. Substituting names of objects of an actual schema for these abstract names provides fully or partially instantiated transformations. For example, ('PHONE','has') ← ATTRIBUTE-to-ET('CUSTOMER','Phone') specifies the transformation of attribute Phone of entity type CUSTOMER, while (EA2,R) ← ATTRIBUTE-to-ET('CUSTOMER',A2) specifies the family of transformations of any attribute of CUSTOMER entity type.

The concept of transformation is valid whatever the granularity of the object it applies to. For instance, transforming conceptual schema CS into equivalent physical schema PS can be modelled as a (complex) semantics-preserving transformation CS-to-PS = <CS-to-PS, cs-to-ps> in such a way that PS = CS-to-PS(CS). This transformation has an inverse, PS-to-CS = <PS-to-CS, ps-to-cs> so that CS = PS-to-CS(PS).

6.3.2 Compound Schema Transformation

A compound transformation $\Sigma = \Sigma_2 \circ \Sigma_1$ is obtained by applying Σ_2 on the database (schema and data) that results from the application of Σ_1 [216]. Most complex database engineering processes, particularly database design and reverse engineering, can be modelled as compound semantics-preserving transformations. For instance, transformation CS-to-PS referred to here above actually is a compound transformation, since it comprises logical design, that transforms a conceptual schema into a logical schema, followed by physical design, that transforms the logical schema into a physical schema [43]. So, the database design process can be modelled by transformation CS-to-PS = LS-to-PS ∘ CS-to-LS, while the reverse engineering process is modelled by PS-to-CS = LS-to-CS ∘ PS-to-LS.

6.3.3 Transformation History and Schema Mapping

The *history* of an engineering process is the formal trace of the transformations that were carried out during its execution. Each transformation is entirely specified by its signature. The sequence of these signatures reflects the order in which the transformations were carried out. The history of a process provides the basis for such operations as undoing and replaying parts of the process. It also supports the traceability of the source and target artefacts.

In particular, it formally and completely defines the mapping between a source schema and its target counterpart when the latter was produced by means of a transformational process. Indeed, the chain of transformations that originates from any definite source object precisely designates the resulting objects in the target schema, as well as the way they were produced. However, the history approach to mapping

specification has proved complex, essentially for three reasons [218]. First, a history includes information that is useless for schema migration. In particular, the signatures often include additional information for undoing and inverting transformations. Second, making histories evolve consistently over time is far from trivial. Third, real histories are not linear, due to the exploratory nature of engineering processes. Therefore, simpler mappings are often preferred, even though they are less powerful. For instance, we proposed the use of the following lighweight technique based on stamp propagation [232]. Each source object receives a unique stamp that is propagated to all objects resulting from the successive transformations. When comparing the source and target schemas, the objects that have the same stamp exhibit a pattern that uniquely identifies the transformation that was applied on the source object. This approach is valid provided that (1) only a limited set of transformations is used and (2) the transformation chain from each source object is short (one or two operations). Fortunately, these conditions are almost always met in real database design.

6.3.4 Program Transformation

Program transformation is a modification or a sequence of modifications applied to a program. Converting a program generally involves basic transformation steps that can be specified by means of *rewrite rules*. Term rewriting is the exhaustive application of a set of rewrite rules to an input term (e.g., a program) until no rule can be applied anywhere in the term. Each rewrite rule uses pattern matching to recognise a subterm to be transformed and replaces it with a target pattern instance.

Program transformations form a sound basis for application program conversion in the context of database migration. Indeed, the legacy I/O statements have to be rewritten with two concerns in mind, namely making the program comply with the new DMS API, and, more important, adapting the program logic to the new schema. The latter adaptation obviously depends on the way the legacy database schema was transformed into the new schema. This issue has already been addressed in previous work [116]. We have proposed a general approach, based on *coupled transformations* [306], according to which program rewrite rules are associated to schema transformations in a DML-independent manner.

For instance, Figure 6.4 shows an abstract rewrite rule that propagates the schema transformation depicted in Figure 6.3 to primitives that create an instance of entity type A from the values of variables $a1$, $a2_1$, ..., $a2_N$, $a3$. Since attribute $A2$ has been converted into an entity type, the way instances of A are created has to be changed. Creating an instance of entity type A now involves the creation of N instances of entity type $EA2$ within an extra loop. Created instances of $EA2$ are connected to instance a of A through relationship type R.

6.4 Schema Conversion

The schema conversion strategies mainly differ in the way they cope with the explicit and implicit constructs (that is, the data structures and the integrity constraints) of the

$$
\begin{array}{l}
create\ a := A((:A1 = a1) \\
\quad and\ (:A2[1] = a2_1) \\
\quad and\ (:A2[2] = a2_2) \\
\quad \cdots \\
\quad and\ (:A2[N] = a2_N) \\
\quad and\ (:A3 = a3))
\end{array}
\quad t_c \rightarrow \quad
\begin{array}{l}
create\ a := A((:A1 = a1) \\
\quad and\ (:A3 = a3)) \\
for\ i\ in\ 1..N\ do \\
\quad create\ ea2 := EA2((:A2 = a2_i) \\
\quad and\ (R : a)) \\
end for
\end{array}
$$

Fig. 6.4. Create mapping t_c associated with structural mapping T of Fig. 6.3

source schema. An *explicit construct* is declared in the DDL code [6] of the schema and can be identified through examination or parsing of this code. An *implicit construct* has not been declared, but, rather, is controlled and managed by external means, such as decoding and validating code fragments scattered throughout the application code. Such construct can only be identified by sophisticated analysis methods exploring the application code, the data, the user interfaces, to mention the most important sources.

The schema conversion process analyses the legacy application to extract the source physical schema (SPS) of the underlying database 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, called the *physical* schema conversion, merely simulates the explicit constructs of the legacy database into the target DMS. According to the second one, the *conceptual* schema conversion, the complete semantics of the legacy database is retrieved and represented into the technology-neutral conceptual schema (CS), which is then used to develop the new database.

6.4.1 Physical Conversion Strategy (D1)

Principle

According to this strategy (Figure 6.5) each explicit construct of the legacy database is directly translated into its closest equivalent in the target DMS. For instance, considering a standard file to SQL conversion, each record type is translated into a table, each top-level field becomes a column and each record/alternate key is translated into a primary/secondary key. No conceptual schema is built, so that the semantics of the data is ignored.

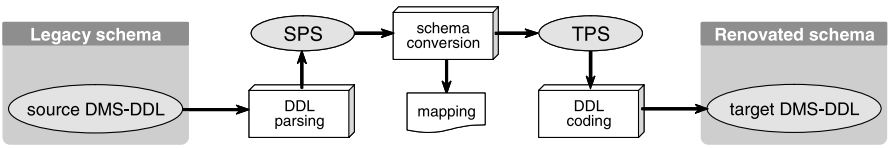

Fig. 6.5. Physical schema conversion strategy (D1)

[6] DDL: Data Description Language.

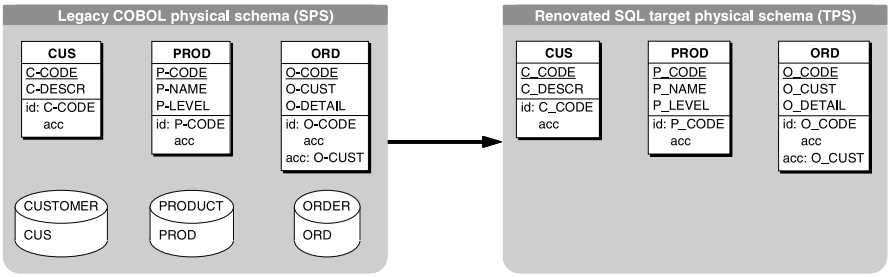

Fig. 6.6. Example of COBOL/SQL physical schema conversion

Methodology

The *DDL parsing* process analyses the DDL code to retrieve the physical schema of the source database (SPS). This schema includes explicit constructs only. It is then converted into its target DMS equivalent (TPS) through a straightforward *one-to-one* mapping and finally coded into the target DDL. The *schema conversion* process also produces the source to target schema mapping.

Illustration

The analysis of the file and record declarations produces the SPS (Figure 6.6/left). Each COBOL record type is translated into an SQL table, each field is converted into a column and object names are made compliant with the SQL syntax (Figure 6.6/right). In this schema, a box represents a physical entity type (record type, table, segment, etc.). The first compartment specifies its name, the second one gives its components (fields, columns, attributes) and the third one declares secondary constructs such as keys and constraints (*id* stands for primary identifier/key, *acc* stands for access key, or index, and *ref* stands for foreign key). A cylinder represents a data repository, commonly called a file.

6.4.2 Conceptual Conversion Strategy (D2)

Principle

This strategy aims at producing a target schema in which all the semantics of the source database are made explicit, even those conveyed by implicit source constructs. In most cases, there is no complete and up to date documentation of the information system, and in particular of the database. Therefore, its logical and conceptual schemas must be recovered before generating the target schema. The physical schema of the legacy database (SPS) is extracted and transformed into a conceptual schema (CS) through reverse engineering. The conceptual schema is then transformed into the physical schema of the target system (TPS) through standard database development techniques.

Methodology

The left part of Figure 6.7 depicts the three steps of a simplified database reverse engineering methodology used to recover the logical and conceptual schemas of the source database.

- As in the first strategy, the first step is the parsing of the DDL code to extract the physical schema (SPS), which only includes the explicit constructs.
- The *schema refinement* step consists in refining the SPS by adding the implicit constructs that are identified through the analysis of additional information sources, such as the source code of the application programs and the database contents, to mention the most common ones. Program code analysis performs an in-depth inspection of the way the programs use and manage the data. Data validation, data modification and data access programming *clichés* are searched for in particular, since they concentrate the procedural logic strongly linked with data properties. The existing data are also analysed through data mining techniques, either to detect constraints, or to confirm or discard hypotheses on the existence of constraints. This step results in the source logical schema (SLS), that includes the explicit representation of such constructs as record and field decomposition, uniqueness constraints, foreign keys or enumerated domains that were absent in SPS. The history SPS-to-SLS of the refinement process forms the first part of the source-to-target mapping.
- The final step is *schema conceptualisation* that semantically interprets the logical schema. The result is expressed by the conceptual schema (CS). This schema is technology independent, and therefore independent of both the legacy and new DMSs. The history SLS-to-CS of this process is appended to the source-to-target mapping.

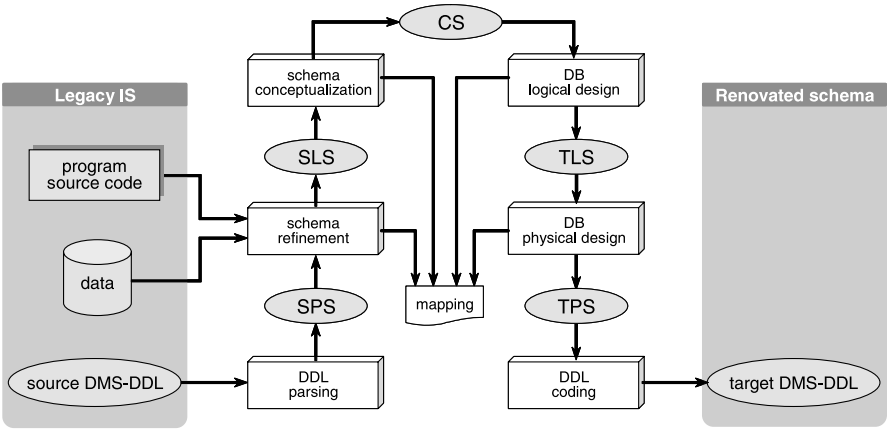

Fig. 6.7. Conceptual schema conversion strategy (D2)

A complete presentation of this reverse engineering methodology can be found in [215] and [214], together with a fairly comprehensive bibliography on database reverse engineering.

The conceptual schema is then transformed into an equivalent logical schema (TLS), which in turn is transformed into the physical schema (TPS). TPS is then used to generate the DDL code of the target database. These processes are quite standard and are represented in the right part of Figure 6.7. The histories CS-to-TLS and TLS-to-TPS are added to the source-to-target mapping. The mapping SPS-to-TPS is now complete, and is defined as SPS-to-SLS ∘ SLS-to-CS ∘ CS-to-TLS ∘ TLS-to-TPS.

Illustration

The details of this reverse engineering case study have been described in [219]. We sketch its main steps in the following. The legacy physical schema SPS is extracted as in the first approach (Figure 6.8/top-left).

The *Refinement* process enriches this schema with the following implicit constructs:

(1) Field `O-DETAIL` appears to be compound and multivalued, thanks to program analysis techniques based on variable dependency graphs and program slicing.

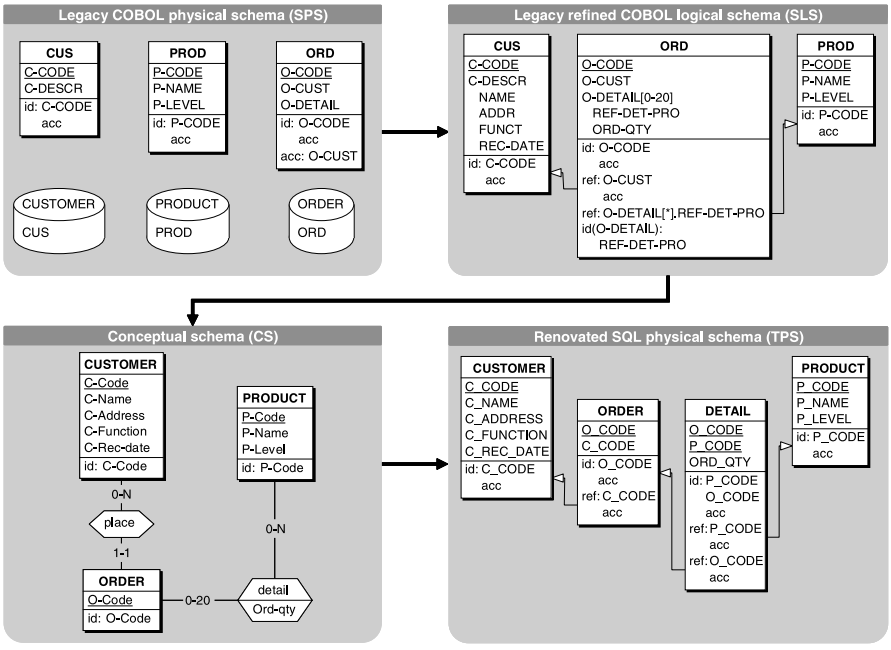

Fig. 6.8. Example of COBOL/SQL conceptual schema conversion

(2) The implicit foreign keys `O-CUST` and `REF-DET-PRO` are identified by schema names and structure patterns analysis, program code analysis and data analysis.
(3) The multivalued identifier (uniqueness constraint) `REF-DET-PRO` of `O-DETAIL` can be recovered through the same techniques.

The resulting logical schema SLS is depicted in Figure 6.8/top-right.

During the *data structure conceptualisation*, the implementation objects (record types, fields, foreign keys, arrays,...) are transformed into their conceptual equivalent to produce the conceptual schema CS (Figure 6.8/bottom-left).

Then, 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 6.8.bottom-right) and the code of the new database is generated.

6.5 Data Conversion

6.5.1 Principle

Data conversion is handled by a so-called Extract-Transform-Load (ETL) processor (Figure 6.9), which transforms the data from the data source to the format defined by the target schema. Data conversion requires three steps. First, it performs the extraction of the data from the legacy database. Then, it transforms these data in such a way that their structures match the target format. Finally, it writes these data in the target database.

Data conversion relies on the mapping that holds between the source and target physical schemas. This mapping is derived from the instance mappings (t) of the source-to-target transformations stored in the history.

Deriving data conversion from the physical schema conversion (D1) is straightforward. Indeed, both physical schemas are as similar as their DMS models permit, so that the transformation step most often consists in data format conversion.

The conceptual schema conversion strategy (D2) recovers the conceptual schema (CS) and the target physical schema (TPS) implements all the constraints of this schema. Generally, both CS and TPS include constraints that are missing in SPS, and that the source data may violate. Thus data migration must include a preliminary data cleaning step that fixes or discards the data that cannot be loaded in the target database [423]. This step cannot always be automated. However, the schema refinement step identifies all the implicit constraints and produces a formal specification

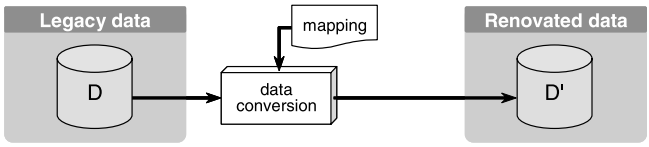

Fig. 6.9. Data migration architecture: converter and schema transformation

for the data cleaning process. It must be noted that the physical schema conversion strategy (D1) makes such data cleaning useless. Indeed, both SPS and TPS express the same constraints that the source data are guaranteed to satisfy.

6.5.2 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 by $<T,t>$, the instance mapping sps-to-tps is automatically derived from the compound transformation SPS-to-TPS built in the schema conversion process. The converter is based on the structural mappings SPS-to-TPS to write the extraction and insertion requests and on the corresponding instance mappings sps-to-tps for data transformation.

6.6 Program Conversion

The program conversion process aims at re-establishing the consistency that holds between application programs and the migrated database. The nature of this consistency is twofold. First, the programs have to comply with the API of the DMS, by using the right data manipulation language and interaction protocols. Second, the programs have to manipulate the data in their correct format, i.e., the format declared in the database schema.

This section analyses the three program modification strategies specified in Figure 6.2. The first one relies on *wrapper technology* (P1) to map the access primitives onto the new database through wrapper invocations that replace the DML statements of the legacy DMS. The second strategy (P2) replaces each statement with its equivalent in the new DMS-DML. According to the P3 strategy, the access logic is rewritten to comply with the DML of the new DMS. In strategies P2 and P3, access statements are expressed in the DML of the new DMS.

In order to compare the three program conversion strategies, we will apply them successively on the same legacy COBOL fragment, given in Figure 6.10. This code fragment deletes all the orders placed by a given customer.

6.6.1 Wrapper Strategy (P1)

Principle

In migration and interoperability architectures, wrappers are popular components that convert legacy interfaces into modern ones. Such wrappers allow the reuse of legacy components [464] (e.g., allow Java programs to access COBOL files). The

```
DELETE-CUS-ORD.
   MOVE C-CODE TO O-CUST.
   MOVE 0 TO END-FILE.
   READ ORDERS KEY IS O-CUST
     INVALID KEY MOVE 1 TO END-FILE.
   PERFORM DELETE-ORDER UNTIL END-FILE = 1.

DELETE-ORDER.
   DELETE ORDERS.
   READ ORDERS NEXT
     AT END MOVE 1 TO END-FILE
     NOT AT END
       IF O-CUST NOT = C-CODE
          MOVE 1 TO END-FILE.
```

Fig. 6.10. A legacy COBOL code fragment that deletes the orders corresponding to a given customer

wrappers discussed in this chapter are of a different nature, in that they simulate the legacy data interface on top of the new database. For instance, they allow COBOL programs to *read*, *write*, *rewrite* records that are built from rows extracted from a relational database. In a certain sense, they could be called *backward* wrappers. An in-depth analysis of both kinds of wrappers can be found in [497].

The wrapper conversion strategy attempts to preserve the logic of the legacy programs and to map it on the new DMS technology [84]. A *data wrapper* is a *data model conversion* component that is called by the application program to carry out operations on the database. In this way, the application program invokes the wrapper instead of the legacy DMS. If the wrapper simulates the modelling paradigm of the legacy DMS and its interface, the alteration of the legacy code is minimal. It mainly consists in replacing 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, converts them to the appropriate legacy format [409] (Figure 6.11) and delivers them to the application program.

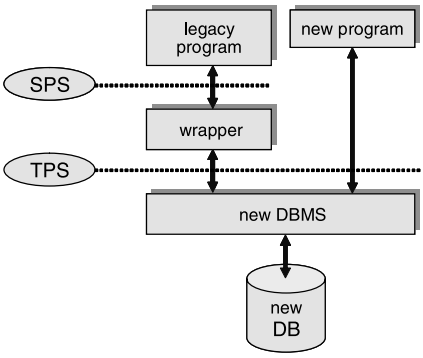

Fig. 6.11. Wrapper-based migration architecture: a wrapper allows the data managed by a new DMS to be accessed by the legacy programs

Methodology

Schemas SPS and TPS, as well as the mapping between them (SPS-to-TPS) provide the necessary information to derive the procedural code of the wrappers. For each COBOL source record type, a wrapper is built that simulates the COBOL file handling statements. The simulated behaviour must also include the management of currency indicators (internal dynamic pointers to current records) as well as error handling.

Once the wrappers have been built, they have to be interfaced with the legacy programs. This can be done by replacing, in the latter, original data access operations with wrapper invocations. Such a transformation is straightforward, each instruction being replaced with a call to the corresponding wrapper and, in some cases, an additional test. In the case of COBOL file handling, the test checks the value of the wrapper status in order to simulate *invalid key* and *at end* clauses.

Legacy code adaptation also requires other minor reorganizations like modifying the *environment division* and the *data division* of the programs. The declaration of files in the *environment division* can be discarded. The declaration of record types has to be moved from the *input-output section* to the *working storage section*. The declarations of new variables used to call the wrapper (action, option and status) are added to the *working storage section*. Finally, new code sections are introduced into the program (e.g., database connection code).

Some legacy DMS, such as MicroFocus COBOL, provide an elegant way to interface wrappers with legacy programs. They allow programmers to replace the standard file management library with a customised library (the *wrapper*). In this case, the legacy code does not need to be modified at all.

The <D1,P1> and <D2,P1> strategies only differ in the complexity of the wrappers that have to be generated. The program transformation is the same in both strategies since each legacy DML instruction is replaced with a wrapper invocation. The code of the wrappers for the <D1,P1> strategy is trivial because each explicit data structure of the legacy database is directly translated into a similar structure of the target database. In the <D2,P1> strategy the conceptual schema is recovered and the new physical schema can be very different from the legacy one. For instance, a record can be split into two or more tables, a table may contain data from more than one record, new constraints might be implemented into the new DMS, etc. In this strategy, translating a `READ` command may require to access more than one table and to perform additional tests and loops.

Illustration

To illustrate the way data wrappers are used, let us consider the legacy COBOL fragment of Figure 6.10, which comprises `READ` and `DELETE` primitives. As shown in Figure 6.12, each primitive is simply replaced with a corresponding wrapper invocation. From the program side, the wrapper is a black box that simulates the behaviour of the COBOL file handling primitives on top of the SQL database. Note that the P1 program adaptation strategy does not depend on the schema conversion strategy.

```
       DELETE-CUS-ORD.
          MOVE C-CODE TO O-CUST.
          MOVE 0 TO END-FILE.
          SET WR-ACTION-READ TO TRUE.
          MOVE "KEY IS O-CUST" TO WR-OPTION.
          CALL WR-ORDERS USING WR-ACTION, ORD, WR-OPTION, WR-STATUS
          IF WR-STATUS-INVALID-KEY MOVE 1 TO END-FILE.
          PERFORM DELETE-ORDER UNTIL END-FILE = 1.

       DELETE-ORDER.
          SET WR-ACTION-DELETE TO TRUE.
          CALL WR-ORDERS USING WR-ACTION, ORD, WR-OPTION, WR-STATUS.
          SET WR-ACTION-READ TO TRUE.
          MOVE "NEXT" TO WR-OPTION.
          CALL WR-ORDERS USING WR-ACTION, ORD, WR-OPTION, WR-STATUS.
          IF WR-STATUS-AT-END
             MOVE 1 TO END-FILE
          ELSE
             IF O-CUST NOT = C-CODE
                MOVE 1 TO END-FILE.
```

Fig. 6.12. Code fragment of Fig. 6.10 converted using the *Wrapper* strategy (P1)

This choice only affects the complexity of the wrapper code, since the latter is directly derived from the mapping that holds between the legacy and new database schemas.

6.6.2 Statement Rewriting (P2)

Principle

This program modification technique depends on the schema conversion strategy. It consists in replacing legacy DMS-DML statements with native DML statements of the new DMS. For example, every file access statement in a COBOL program has to be replaced with an equivalent sequence of relational statements. As for the wrapper strategy, program data structures are left unchanged. Consequently, the relational data must be stored into the legacy COBOL variables.

In the case of the physical schema conversion strategy (D1), the conversion process can be easily automated, thanks to the simple SPS-to-TPS mapping. The conceptual schema conversion strategy (D2) typically flattens complex COBOL structures in the target relational schema. This makes the use of additional loops necessary when retrieving the value of a compound multivalued COBOL variable. Although the substitution process is more complex than in the D1 strategy, it can also be fully automated.

Methodology

The program modification process may be technically complex, but does not need sophisticated methodology. Each DML statement has to be located, its parameters have to be identified and the new sequence of DML statements has to 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 exist several techniques to express the typical `START/READ NEXT` loop with SQL 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` or `READ KEY IS` initial statement, making it impossible to identify the record key used. A description of a specific technique that solves this problem is provided below.

Illustration

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

The technique illustrated in Figure 6.13 solves this problem. This technique 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 example of Figure 6.13 shows the <D2,P2> conversion the COBOL code fragment of Figure 6.10. During the schema conversion process, the `O-DETAIL` compound multivalued field has been converted into the `DETAIL` SQL table. So, rebuilding the value of `O-DETAIL` requires the execution of a loop and a new `FILL-ORD-DETAIL` procedure. This new loop retrieves the details corresponding to the current `ORD` record, using a dedicated SQL cursor.

6.6.3 Logic Rewriting (P3)

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 the program logic. For example, the processing code of a COBOL record type may be replaced with 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 that *should* be modified by the programmer and to give hints on how to rewrite them. However, modifying the code is still up to the programmer.

```
EXEC SQL DECLARE CURSOR ORD_GE_K1 FOR
  SELECT CODE, CUS_CODE
  FROM ORDERS WHERE CUS_CODE >= :O-CUST
  ORDER BY CUS_CODE
END-EXEC.
...
EXEC SQL DECLARE CURSOR ORD_DETAIL FOR
  SELECT PROD_CODE, QUANTITY
  FROM DETAIL WHERE ORD_CODE = :O-CODE
END-EXEC.
...
DELETE-CUS-ORD.
  MOVE C-CODE TO O-CUST.
  MOVE 0 TO END-FILE.
  EXEC SQL
    SELECT COUNT(*) INTO :COUNTER
    FROM ORDERS WHERE CUS_CODE = :O-CUST
  END-EXEC.
  IF COUNTER = 0
     MOVE 1 TO END-FILE
  ELSE
     EXEC SQL OPEN ORD_GE_K1 END-EXEC
     MOVE "ORD_GE_K1" TO ORD-SEQ
     EXEC SQL
       FETCH ORD_GE_K1
       INTO :O-CODE, :O-CUST
     END-EXEC
     IF SQLCODE NOT = 0
        MOVE 1 TO END-FILE
     ELSE
        EXEC SQL OPEN ORD_DETAIL END-EXEC
        SET IND-DET TO 1
        MOVE 0 TO END-DETAIL
        PERFORM FILL-ORD-DETAIL UNTIL END-DETAIL = 1
     END-IF
  END-IF.
  PERFORM DELETE-ORDER UNTIL END-FILE = 1.
DELETE-ORDER.
  EXEC SQL
     DELETE FROM ORDERS
     WHERE CODE = :O-CODE
  END-EXEC.
  IF ORD-SEQ = "ORD_GE_K1"
     EXEC SQL
       FETCH ORD_GE_K1 INTO :O-CODE,:O-CUST
     END-EXEC
  ELSE IF ...
    ...
  END-IF.
  IF SQLCODE NOT = 0
     MOVE 1 TO END-FILE
  ELSE
     IF O-CUST NOT = C-CODE
        MOVE 1 TO END-FILE.
...
FILL-ORD-DETAIL SECTION.
  EXEC SQL
     FETCH ORD_DETAIL
     INTO :REF-DET-PRO(IND-DET),:ORD-QTY(IND-DET)
  END-EXEC.
  SET IND-DET UP BY 1.
  IF SQLCODE NOT = 0
     MOVE 1 TO END-DETAIL.
```

Fig. 6.13. Code fragment of Fig. 6.10 converted using the *Statement Rewriting* strategy (P2)

This strategy can be justified if the whole system, that is database and programs, has be renovated in the long term (strategy <D2,P3>). After the reengineering, the new database and the new programs take advantage of the expressiveness of the new technology. When the new database is just a *one-to-one* translation of the legacy database (<D1,P3>), this strategy can be very expensive for a poor result. The new database just simulates the old one and takes no advantage of the new DMS. Worse, it inherits all the flaws of the old database (bad design, design deteriorated by maintenance, poor expressiveness, etc.). Thus, we only address the <D2,P3> strategy in the remaining of this section.

Methodology

The P3 strategy is much more complex than the previous ones 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 and understanding the statements and the data objects that depend on these access statements and (3) rewriting these statements as a whole and redefining these data objects.

Illustration

Figure 6.14 shows the code fragment of Figure 6.10 converted using the *Logic Rewriting* strategy. The resulting code benefits from the full power of SQL. The two-step *position then delete* pattern, which is typical of navigational DMS, can be replaced with a single predicate-based *delete* statement.

```
       DELETE-CUS-ORD.
          EXEC SQL
            DELETE FROM ORDERS
            WHERE CUS_CODE = :C-CODE
          END-EXEC.
          IF SQLCODE NOT = 0 THEN GO TO ERR-DEL-ORD.
```

Fig. 6.14. Code fragment of Fig. 6.10 converted using the *Logic Rewriting* strategy (P3)

6.7 Tool Support

Some of the information system migration strategies we developed in this chapter have been implemented using two complementary transformational technologies, namely DB-MAIN and the ASF+SDF Meta-Environment.

6.7.1 The Tools

The DB-MAIN CASE Environment

DB-MAIN [143] is a data-oriented CASE environment developed by the Laboratory of Database Application Engineering (LIBD) of the University of Namur. Its purpose is to help the analyst in the design, reverse engineering, reengineering, maintenance and evolution of database applications.

DB-MAIN offers general functions and components that allow the development of sophisticated processors supporting data-centred application renovation:

- A generic model of schema representation based on the GER (Generic Entity/Relationship) model to describe data structures in all abstraction levels and according to all popular modelling paradigms.
- A graphical interface to view the repository and apply operations.
- A transformational toolbox rich enough to encompass most database engineering and reverse engineering processes.
- Customizable assistants (e.g., transformation, reverse engineering, conformity analysis) to help solve complex and repetitive problems.
- A history processor to record, replay, save or invert history.

DB-MAIN also includes several processors specific to the reverse engineering process [229], such as DDL parsers for most popular DMSs, a foreign key discovery assistant, and program analysis tools (pattern matching, variable dependency analysis and program slicing). Experience of actual reverse engineering taught us that there are no two reengineering projects are the same. Hence the need for programmable, extensible and customisable tools. DB-MAIN (and more specifically its meta functions) includes features to extend its repository and develop new functions. It includes in particular a 4GL (*Voyager2*) as well as a Java API that allow analysts to quickly develop their own customised processors [215].

The ASF+SDF Meta-Environment

The ASF+SDF Meta-Environment [515] is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language. It is developed by the SEN1 research group of the CWI in Amsterdam. In the context of system migration, the ASF+SDF Meta-Environment provides tool generators to support the program conversion step. It allows both defining the syntax of programming languages and specifying transformations of programs written in such programming languages [514].

The next sections describe the tool support in the different steps of the methodologies described in this chapter for schema, data and program conversion.

6.7.2 Schema Conversion

The physical schema conversion strategy uses simple tools only, such as a DDL parser to extract SPS, an elementary schema converter to transform SPS into TPS and a DDL generator. Complex analysers are not required.

In the conceptual schema conversion strategy, extracting SPS and storing it in the CASE tool repository is done through a DDL parser (SQL, COBOL, IMS, CODASYL, RPG, XML) from the parser library. Schema refinement requires schema, data and program analysers. Data structure conceptualization and database design are based on schema transformations. Code generators produce the DDL code of the new database according to the specifications of TPS.

6.7.3 Mapping Definition

We use the transformation toolkit of DB-MAIN to carry out the chain of schema transformations needed during the schema conversion phase. DB-MAIN automatically generates and maintains a history log of all the transformations that are applied to the legacy DB schema (SPS) to obtain the target DB schema (TPS). This history log is formalised in such a way that it can be analysed and transformed. Particularly, it can be used to derive both the mappings between SPS and TPS. A visual mapping assistant has been developed to support the definition, the visualization and the validation of inter-schema mappings. This tool is based on the stamping technique described in Section 6.3.3.

6.7.4 Data Conversion

Writing data converters manually is an expensive task, particularly for complex mappings (for simple mappings parametric ETL converters are quite sufficient). The DB-MAIN CASE tool includes specific history analysers and converter generators that have been described in [146].

6.7.5 Program Conversion

Wrapper Generation

So far, wrapper generators for COBOL-to-SQL and IDS/II[7]-to-SQL have been developed. These generators are implemented through Java plug-ins of DB-MAIN, and require the following inputs:

- the legacy database schema
- an optional intermediate schema
- the target database schema
- the mapping between these two (three) schemas

The generators produce the code that provides the application programs with a legacy interface to the new database. In practice, we generate one wrapper per legacy record type. Each generated wrapper is a COBOL program with embedded SQL primitives. The generated wrappers simulate the legacy DMS on top on the renovated database. Note that the same tools can be used for supporting both P1 and P2 program conversion strategies, which mainly differ from the target location of the generated code (wrapper or new program section).

[7] IDS/II is the BULL implementation of CODASYL.

Legacy Code Transformation

The adaptation of the legacy application programs relies on the ASF+SDF Meta-Environment. We use an SDF version of the IBM VS COBOL II grammar, which was obtained by Lämmel and Verhoef [304]. We specify a set of rewrite rules (ASF equations) on top of this grammar to obtain two similar program transformation tools. The first tool is used in the context of COBOL-to-SQL migration, while the second one supports IDS/II-to-SQL conversion.

The main input arguments of the program transformers are automatically generated. These parameters include:

- the list of the migrated record types
- additional variable declarations
- additional program code sections
- owner and members of each set (IDS/II)
- list of the declared record keys (IDS/II)

The program transformation tools are suitable in case of partial migration, i.e., when only some legacy record types actually are migrated to the new database platform. In that case, only the DML instructions manipulating migrated data are adapted. The other DML instructions, which still access the legacy data, are left unchanged.

6.8 Industrial Application

We have been involved in several industrial reverse engineering and reengineering projects during the last three years. In this section, we particularly report on an ongoing IDS/II-to-SQL database migration project.

6.8.1 Project Overview

The project aims at migrating a large COBOL system towards a relational (DB2) database platform. The legacy system runs on a Bull mainframe and is made of nearly 2300 programs, totalling more than 2 million lines of COBOL code. The information system makes use of an IDS/II database. The source physical DB schema comprises 231 record types, 213 sets and 648 fields. The migration strategy chosen is based on the combination of a conceptual database conversion (D2) and a wrapper-based program conversion (P1).

6.8.2 Process Followed

The project started with a prototyping phase, during which a consistent subset of the data and programs has been fully migrated. This initial phase aims at verifying the correctness of the overall migration through a systematic testing process. The database subset includes 26 IDS/II record types and 31 sets. The legacy programs selected for conversion comprise 51 KLOC and make use of almost every possible

IDS/II statement (*find*, *get*, *erase*, *store*, *modify*, *connect*, *disconnect*, etc.). The tests, performed with the help of IDS/II experts from the customer side, have shown the correctness of the automated program conversion.

Below, we describe the main phases that we followed to migrate the complete legacy system.

Inventory

The purpose of the inventory process is twofold. First, it aims at checking that we have received a complete and consistent set of source code files from the customer. Second, it allows us to get a rapid overview of the application architecture in order to evaluate the complexity of the migration task, as well as the part of the work that cannot be automated. In this project, the inventory phase produced the following results :

- complete statistics about the IDS/II statements (number, type, location);
- the program call graph, specifying which program calls which program;
- the database usage graph, specifying which program uses which IDS/II record type;
- a classification of the legacy source code files based on their database usage (no access, indirect access or direct access).

Schema Conversion Through DBRE

During the database reverse engineering process, program analysis techniques have been used in order to retrieve different kinds of information about the legacy database. In particular, dataflow analysis allowed us to find which program variables are used to manipulate the records, in order to deduce a more precise record decomposition. Dataflow analysis was also used to elicit implicit data dependencies that exist between database fields, among which potential foreign keys. Our dataflow analysis approach is inspired by the interprocedural slicing algorithm proposed by Horwitz et al. [235], based on the system dependency graph (SDG). We refer to [117] for more details on the use of SDGs in database reverse engineering.

Among others, the DBRE process allowed us to:

- recover finer-grained structural decompositions for record types and attributes;
- retrieve implicit data dependencies, including 89 foreign keys, 37 computed foreign keys, and 60 other redundancies.

Table 6.1 gives a comparison of the successive versions of the database schema. The physical IDS/II schema is the initial schema extracted from the DDL code (here we consider the subset of the schema actually migrated). The refined IDS/II schema is the physical schema with a finer-grained structure. It was obtained by resolving numerous copybooks in which structural decompositions of physical attributes are declared. In the refined IDS schema, most attributes are declared several times through *redefines* clauses, hence the huge total number of attributes. The conceptual schema

Table 6.1. Comparison of successive versions of the complete database schema

	Physical IDS/II	Refined IDS/II	Conceptual	Relational DB2
# entity types	159	159	156	171
# relationship types	148	148	90	0
# attributes	458	9 027	2 176	2 118
max # att./entity type	8	104	61	94

is the result of the conceptualization phase. It comprises only one declaration per attribute. When a conflict occurs, the chosen attribute decomposition is the one the analyst considers to be the most expressive. In addition, the number of entity type is different since some technical record types were discarded while other ones were split (sub-types). Finally, the relational schema shows an increase in the number of entity types, due to the decomposition of arrays, as well as a reduction of the number of attributes due to the aggregation of compound fields.

Data Validation and Migration

During the schema conversion phase, the mapping of the various components is recorded between the successive schemas, such that we know precisely how each concept is represented in each schema. From such mappings we can generate two kinds of programs:

- Data validators, which check if the legacy data comply with all recovered implicit constraints;
- Data migrators, that actually migrate the legacy data to the relational database.

The data validation step revealed that many implicit referential constraints were actually violated by the legacy data. This is explained by the fact that most rules are simply encoding rules which are not always checked again when data are updated, and by the fact that users find tricks to bypass some rules.

Wrapper-Based Program Conversion

The wrapper generation phase produced 159 database wrappers. Each generated wrapper is a COBOL program containing embedded SQL primitives. The total wrapper code size is about 450 KLOC.

The results obtained during the legacy code adaptation are summarised in Table 6.2. A total of 669 programs and 3 917 copybooks were converted. We notice that about 92% of the IDS/II verbs were transformed automatically, while the manual work concerned 85 distinct source code files only.

Table 6.2. Program transformation results

	Migrated	Manually transformed
# programs	669	17
# copybooks	3 917	68
# IDS/II verbs	5 314	420

6.8.3 Lessons Learned

Methodology

As in previous projects, the initial inventory step proved to be critical. It required several iterations since we discovered missing copybooks and programs, as well as code fragments containing syntax errors. The prototyping phase also proved valuable, since it allowed us to detect problems early in the process and to better confront our results with the customer requirements. Another conclusion is that the database reverse engineering process may benefit from the data validation phase. Indeed, analysing database contents does not only allow to detect errors, it may also serve as a basis for formulating new hypotheses about potential implicit constraints.

Automation

Although large-scale system conversion needs to be supported by scalable tools, the full automation of the process is clearly unrealistic. Indeed, such a project typically requires several iterations as well as multiple human decisions. In particular, while previous smaller projects allowed us to automate the schema design process with minor manual corrections, assisted manual conversion becomes necessary when dealing with larger schemas. For instance, translating a compound attribute into SQL columns can be done either by disaggregation, by extraction or by aggregation. In this project, the chosen technique depended on the nature of the compound attribute (e.g., each compound attribute representing a date has been translated as a single column). The database design must respect various other constraints like the type and naming conventions of the customer.

Wrapper development

Writing correct wrapper generators requires a very good knowledge of the legacy DMS. In this project, the difficulties of wrapper generation were due to the paradigm mismatch between network and relational database systems. Simulating IDS/II verbs on top of a native relational database appeared much more complicated than expected. The generated wrappers must precisely simulate the IDS/II primitives behaviour, which includes the synchronised management of multiple currency indicators, reading sequence orders and returning status codes. Another challenge, as for the data extractors, was to correctly manage IDS/II records that have been split into several SQL tables.

6.9 Strategies Comparison

Six representative strategies of information system migration have been identified. In this section, we compare them according to each dimension and we suggest possible applications for each system migration strategy.

6.9.1 Database Conversion Strategies

The *physical schema conversion* (D1) does not recover the semantics of the database but blindly translates in the target technology the design flaws as well as the technical structures peculiar to the source technology. This strategy can be fully automated, and can be performed manually, at least for small to medium size databases. Further attempts to modify the structure of the database (e.g., adding some fields or changing constraints) will force the analyst to think in terms of the legacy data structures, and therefore to recover their semantics. The source database was optimised for the legacy DMS, and translating it in the new technology most often leads to poor performance and limited capabilities. For example, a COBOL record that includes an array will be transformed into a table in which the array is translated into an unstructured column, making it impossible to query its contents. Doing so would require writing specific programs that recover the implicit structure of the column. Clearly, this strategy is very cheap (and therefore very popular), but leads to poor results that will make future maintenance expensive and unsafe. In particular, developing new applications is almost impossible.

Nevertheless, we must mention an infrequent situation for which this strategy can be valuable, that is, when the legacy database has been designed and implemented in a disciplined way according to the database theory. For instance, a database made up of a collection of 3NF [8] record types can be migrated in a straightforward way to an equivalent relational database of good quality.

The *conceptual schema conversion* (D2) produces a high quality conceptual schema that explicitly represents all the semantics of the data, but from which technology and performance dependent constructs have been discarded. It has also been cleaned from the design flaws introduced by inexperienced designers and by decades of incremental maintenance. This conceptual schema is used to produce the TPS that can use all the expressiveness of the new DMS model and can be optimised for this DMS. Since the new database schema is normalised and fully documented, its maintenance and evolution is particularly easy and safe. In addition, making implicit constraints explicit automatically induces drastic data validation during data migration, and increases the quality of these data. However, this strategy requires a complex reverse engineering process that can prove expensive. For example, the complete reverse engineering of a medium size database typically costs two to four man-months.

[8] 3NF stands for *third normal form.*

6.9.2 Program Conversion Strategies

The *wrapper strategy* (P1) does not alter the logic of the legacy application program. When working on the external data, the transformed program simply invokes the wrapper instead of the legacy DMS primitives. The transformation of the program is quite straightforward: each legacy DMS-DML is replaced with a call to the wrapper. So, this transformation can easily be automated. The resulting program has almost the same code as the source program, so a programmer who has mastered the latter can still maintain the new version without any additional effort or documentation. When the structure of the database evolves, only the wrapper need be modified, while the application program can be left unchanged. The complexity of the wrapper depends on the strategy used to migrate the database. In the D1 strategy, the wrapper is quite simple: it reads one line of the table, converts the column values and produces a record. In the D2 strategy, the wrapper can be very complex, since reading one record may require complex joins and loops to retrieve all the data. Despite the potentially complex mapping between SPS and TPS, which is completely encapsulated into the wrapper, the latter can be produced automatically, as shown in [16]. A wrapper may induce computing and I/O overhead compared to P2 and P3 strategies.

The *statement rewriting* strategy (P2) also preserves the logic of the legacy program but it replaces each legacy DMS-DML primitive statement with its equivalent in the target DMS-DML. Each legacy DMS-DML instruction is replaced with several lines of code that may comprise tests, loops and procedure calls. In our case study the number of lines increased from 390 to almost 1000 when we applied the <D1,P2> strategy. The transformed program becomes difficult to read and to maintain because the legacy code is obscured by the newly added code. If the code must be modified, the programmer must understand how the program was transformed to write correct code to access the database. When the structure of the database is modified, the entire program must be walked through to change the database manipulation statements. In summary, this technique is inexpensive but degrades the quality of the code. In addition, it is fairly easy to automate. As expected, this migration technique is widely used, most often in the <D1,P2> combination.

The *logic rewriting* strategy (P3) changes the logic of the legacy program to explicitly access the new database and to use the expressiveness of the new DMS-DML. This rewriting task is complex and cannot be automated easily. The programmer that performs it must have an in-depth understanding of the legacy database, of the new database and of the legacy program. This strategy produces a completely renovated program that will be easy to maintain at least as far as database logic is concerned.

6.9.3 System Migration Strategies

By combining both dimensions, we describe below typical applications for each of the strategies that have been described.

- **<D1,P1>**: This approach produces a (generally) badly structured database that will suffer from poor performance but preserves the program logic, notably

because the database interface is encapsulated in the wrapper. It can be recommended when the migration must be completed in a very short time, e.g., when the legacy environment is no longer available. Developing new applications should be delayed until the correct database is available. This approach can be a nice first step to a better architecture such as that produced by <D2,P1>. However, if the legacy database already is in 3NF, the result is close to that of strategy <D2,P1>.

- **<D2,P1>**: This strategy produces a good quality database while preserving the program logic. New quality applications can be developed on this database. The legacy programs can be renovated later on, step by step. Depending on the impedance mismatch between the legacy and target technologies, performance penalty can be experienced. For instance, wrappers that simulate CODASYL DML on top of a relational database have to synchronise two different data manipulation paradigms, a process that may lead to significant data access overhead.
- **<D1,P2>**: Despite its popularity, due to its low cost, this approach clearly is the worst one. It produces a database structure that is more obscure than the source one, and that provides poorer performance. The programs are inflated with obscure data management code that makes them complex and more difficult to read, understand and maintain. Such a renovated system cannot evolve at sustainable cost, and therefore has no future. If the legacy database already is in 3NF, the result may be similar to that of strategy <D2,P2>.
- **<D2,P2>**: Produces a good quality database, but the programs can be unreadable and difficult to maintain. It can be considered if no maintenance of the application is planned and the programs are to be rewritten in the near future. If the wrapper overhead is acceptable, the<D2,P1> strategy should be preferred.
- **<D1,P3>**: Data migration produces a very poor quality database that simulates the legacy database. Adapting, at high cost, the program to these awkward structures is meaningless, so that we can consider this strategy not pertinent
- **<D2,P3>**: This strategy provides both a database and a set of renovated programs of high quality, at least as far as database logic is concerned. Its cost also is the highest. This is a good solution if the legacy program language is kept and if the programs have a clean and clear structure.

6.10 Conclusions

The variety in corporate requirements, as far as system reengineering is concerned, naturally leads to a wide spectrum of migration strategies. This chapter has identified two main independent lines of decision, the first one related to the precision of database conversion (schema and contents) and the second one related to program conversion. From them, we were able to identify and analyse six reference system migration strategies. The thorough development of these technical aspects is the major contribution of this chapter since most of these aspects have only been sketched in the literature [84].

Despite the fact that a supporting technology has been developed, and therefore makes some sophisticated strategies realistic at an industrial level, we still lack sufficient experience to suggest application rules according to the global corporate strategy and to intrinsic properties of the legacy system. As is now widely accepted in maintenance, specific metrics must be identified to score the system against typical reference patterns. Such criteria as the complexity of the database schema, the proportion of implicit constructs, the underlying technology, the normalisation level or the redundancy rate, to mention only a few, should certainly affect the feasibility of each migration strategy. Corporate requirements like performance, early availability of components of the renovated system, program independence against the database structure, evolvability, skill of the development team, or availability of human resources are all aspects that could make some strategies more valuable than others.

Though some conclusions could seem obvious at first glance, such as, strategy <D2,P3> yields better quality results than strategy <D1,P2>, we have resisted providing any kind of decision table that would have been scientifically questionable. Indeed, each strategy has its privileged application domains, the identification of which would require much more analysis than we have provided in this chapter. One important lesson we learned in this study is that the quality of the target database is central in a renovated system, and is a major factor in the quality of the programs, whatever the program transformation strategy adopted. For instance, renovated program performance, maintenance costs and the readability of the programs to be developed are strongly dependent on the quality of the database schema.

So far, we have developed a solid methodology and a sophisticated CASE environment for database reverse engineering, wrapper development and automated program conversion (according to P1 and P2 strategies). We have also built a toolset of code analysers, such as a pattern matching engine, a dependency and data flow diagram analyser and a program slicer. They allow us to find code sections that meet structural criteria such as data access sections or the statement streams that influence the state of objects at some point of a program aka program slice).

At present time, we are exploring the automation of the P3 program conversion strategy (*Logic Rewriting*). This strategy aims at adapting the logic of the legacy program to explicitly access the new database and to use the expressiveness of the new DMS-DML. This rewriting task is complex and could not be fully automated. Only the identification of the file access statements and the statements and data objects that depend on them can be automated. These identification tasks relate to the program understanding realm, where such techniques as searching for *clichés*, variable dependency analysis and program slicing (see [538, 229]) are often favourite weapons.

Acknowledgement. Anthony Cleve received support from the Belgian *Région Wallonne* and the European Social Fund via the RISTART project.