MASTER IN COMPUTER SCIENCE

Data-centered applications conversion using program transformations

Cleve, Anthony

Award date:
2004

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Data-centered Applications Conversion using Program Transformations

Anthony Cleve

Mémoire présenté en vue de l'obtention
du grade de Maître en Informatique.

Année Académique 2003-2004

Preface

This Master's thesis is not a personal work at all. It is the result of an active collaboration with two research teams from two distinct scientific communities: the LIBD laboratory of the University of Namur and the SEN1 group of the CWI of Amsterdam. The LIBD team is specialized in database engineering, database reengineering and database reverse engineering. The SEN1 group does research in the area of software reverse engineering, software reengineering and software renovation. I would like to acknowledge all the people working in those two teams for providing me a pleasant and fruitful research environment.

In particular, I would like to thank Professor Jean-Luc Hainaut, my promotor, and Professor Arie van Deursen, my CWI supervisor, for their precious ideas, feedback and support. I also express my gratitude to Jean Henrard for his patience and time to answer my questions and to read early versions of this work.

Working at the CWI has been an excellent experience. I would like to thank all the people of the SEN1 group for being so nice colleagues. Many thanks to Paul Klint, Mark van den Brand, Ralf Lämmel, Jurgen Vinju and Magiel Bruntink for the interest they expressed throughout the realization of this project. I also thank Steven Klusener and Niels Veerman, from the Free University of Amsterdam, for their precious collaboration.

Finally, none of this have been possible without the support of my family and friends. Special thanks to Julie for her patience and support. I am particularly grateful to my mother, Marc, Michael and Mathilde for being a so nice family. However, I could not complete this preface without thinking of my father, Antoine, to whom I dedicate this thesis. I am sure he would be proud of me.

Resume

L'un des défis le plus importants aujourd'hui est celui de la reingénierie des systèmes d'information vers des plateformes techniquement avancées. La réingénierie des données consiste à dériver une base de données moderne à partir d'une base de données ancienne et à adapter les composants logiciels en conséquence. On dispose actuellement d'une maîtrise satisfaisante en ce qui concerne la conversion des données d'un SGBD vers un autre. Mais la transposition des programmes reste un problème relativement peu étudié. Ce mémoire s'attache à explorer des solutions à ce problème, dans le cadre particulier de la conversion de fichiers COBOL vers une base de données relationnelle. Il présente deux stratégies de conversion de programmes et propose, pour chacune d'elles, des outils de transformation automatique.

Mots clés: réingénierie des données, transformation de programme, COBOL.

Abstract

One of the most important challenges in software renovation is DBMS substitution. Data reengineering consists of deriving a new database from a legacy database and adapting the software components accordingly. This database migration process comprises three main steps, namely schema conversion, data conversion and program conversion. While converting the legacy schema into the new DMS and migrating the data instances according to the new schema have been studied for long, technical aspects to data-centered program conversion has been neglected by the scientific community. This Master's thesis addresses this problem in the particular context of the conversion of COBOL files into a relational database. It specifies two different program conversion strategies and explores their automation, by using *Program Transformation* tools.

Keywords: data reengineering, program transformation, COBOL.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem Statement

Nowadays, software information systems play a crucial role in many organizations. Often these systems have a very long life. While hardware can be replaced, software systems and the data they process live on for many decades. This can lead to maintenance problems. Software maintenance consists of keeping the software systems up-to-date and responsive to user needs and changing environments. Maintenance of software amounts to 60 to 85% of total software costs in industry [Wie95].

Legacy System When old software systems become inflexible and difficult to maintain, they are called *legacy systems*. Brodie and Stonebraker [BS95] define a *legacy system* as follows:

> *A legacy system is any system that significantly resists modifications and changes. Typically, a legacy system is big, with millions of lines of code, and more than 10 years old.*

On the one hand, the legacy systems are the lifeblood of the companies. They contain the business knowledge and process mission-critical enterprise data. On the other hand, the legacy systems pose important economical problems to their host organizations [WLB+97]. Among them, let us mention:

- The maintenance of the software components is generally highly costly and time consuming due to the lack of documentation. Often the legacy applications are older than the maintenance programmers working one those applications.
- The legacy systems can hardly evolve to provide new functionalities required by the organization.
- They usually run on obsolete hardware which is expensive to maintain and reduces productivity due to its low speed.

- The legacy systems often are isolated in that they do not easily interface with other applications [HHTH02]. Particularly, they do not integrate well with other legacy systems or new systems.
- The legacy systems are typically written in old programming languages that the young programmers do not learn anymore, e.g., COBOL.

Most of the legacy systems are written in COBOL. COBOL, the *Common Business Oriented Language*, is a third-generation language created in 1950s. Today, there is an estimated of 200 billion lines of COBOL code, representing more than 60 percent of the world's business active software [Ulr02, SPL03].

Many companies have to deal with their existing legacy systems. Most of the legacy systems *work*. But they were never designed for today's business environments. We identify two problematic situations. Firstly, the platforms and technologies around which the legacy system has been built might become so obsolete that they are not supported anymore. Keeping their system running around them can be highly risky. Secondly, the organization might require the legacy system to provide new functionalities, but the system significantly resists to these changes. In both situations, the existing data and functionalities of the system do satisfy the business of the company and are to be preserved.

System Migration Several solutions exist to face the lack of modifiability of legacy systems. One of them, maybe the most popular, is the *System Migration*. Migrating a system consists of replacing one or several of the implementation technologies. Typically, the migration moves the applications and the database to a new platform and new technologies, thereby preserving the data and the functionalities of the system.

Two different migration strategies can be chosen. Firstly, the legacy system can be rewritten *from scratch*. This strategy is often undesirable. It implies that the maintenance is duplicated while building the system, and it is also highly risky. Secondly, the system can be migrated by small *incremental* steps. Businesses often choose this second strategy, seeking to maximize their existing investment and preserve valuable business knowledge.

The *Database First* migration method is one of the incremental migration strategies. It involves the initial migration of the legacy data to a modern Data Management System (DMS) and then incrementally migrating the legacy applications. Migrating the data components first can be much more efficient than trying to deal with the whole application. This incremental approach is also less risky since the permanent data structures are generally the most stable part of the application.

Automatic Program Transformation In this thesis, we will focus on the second step of a Database First migration method: the program conversion. Once the legacy database has been migrated, the legacy programs are to be altered in such a way that they access the new database instead of the legacy data. More specifically, the access statements occurring in the programs must be changed accordingly.

The transformation process of the legacy programs depends on the way the database has been previously converted. Furthermore, this process can (and should) be automated. Indeed, manually transforming source code could be very expensive and dangerous, especially for applications with millions of lines of code.

Program transformation is the act of changing one program into another [PTW]. Automatic program transformation can be defined as a set of transformation rules (or rewriting rules) that are applied to the source code.

It is essential that people trust the automatic program transformation. This leads to two main requirements. Firstly, the program transformation must preserve the semantics of the program. We say that the transformed program must be *equivalent* to the initial program. Secondly, the transformation must be done in a traceable way. For instance, the portions of the source code that have been rewritten must be easily identifiable and completely documented.

1.2 Data Reengineering Strategies

Data Reengineering can be defined as the process of deriving a new database from a legacy database and adapting the software components accordingly. Typically, data reengineering comprises three main steps:

1. *Schema conversion*: the legacy data schema (or structure) is translated into an equivalent schema expressed in the new technology.

2. *Data conversion*: the data instances are migrated from the legacy database to the new one. This second step depends on the schema conversion, since the data are to be converted according to the schema transformations.

3. *Program conversion*: the legacy programs are modified so that they access the new database instead of the legacy one. The functionalities, the programming language and the user interface of the programs are kept unchanged. The program transformation step can be a complex process that depends on the two previous steps.

Henrard et. al. [HHTH02] identify six different data reengineering strategies to migrate data-intensive applications from a legacy DMS to a modern DMS. They consider 2 dimensions of the migration: the database dimension (D) and the program dimension (P).

1.2.1 Database dimension (D)

In order to convert the database, two extreme strategies can be chosen. The first one, the *Physical Conversion* (or D1), consists of translating each construct of the source database into the closest constructs of the target DMS. In the Physical conversion, the semantics of the data is ignored. This conversion strategy is cheap, but it leads to a poor quality database. The second strategy, called *Conceptual Conversion* (or

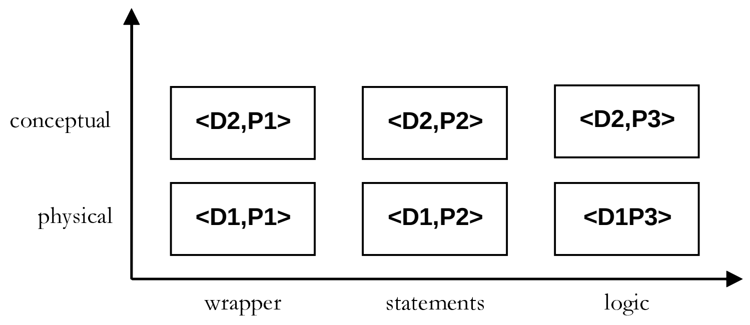


Figure 1.1: *The six reference system migration strategies*

D2), comprises two main steps. First, it recovers the precise semantics description (i.e., the conceptual schema) of the legacy database, through a database reverse engineering (DBRE) phase [Hen03]. Then, it develops the target database from that conceptual schema, by following a standard database methodology. In this way, a good quality and documented database can be obtained. Since the DBRE is often a complex process, the Conceptual conversion can be much more expensive than the Physical conversion.

1.2.2 Program Dimension (P)

Once the database has been converted, the legacy programs are to be altered accordingly. The program conversion consists of transforming the legacy application programs so that they can access the migrated database instead of the legacy database. Three reference strategies can be followed. The first strategy (*Wrapper* or P1) relies on *wrapper* technology to map the access primitives onto the new database. The legacy application programs may now invoke the wrapper instead of the legacy DMS in order to access the migrated data. In the second strategy (*Statement Rewriting* or P2), the access statements are rewritten in order to make them process the new data through the new DMS-DML[1]. These first two strategies do not change the program logic. The purpose of the third strategy (*Logic rewriting* or P3) is to use the full power of the new DMS-DML. This complex conversion process requires a deep understanding of the program logic, since the latter will generally be changed.

The two dimensions define six information system migration strategies, as shown in Figure 1.1. In this Master's thesis, we will focus on two of them: the <D2,P1> strategy and the <D1,P2> strategy.

1.2.3 The <D2,P1> strategy

The Conceptual conversion (D2) Figure 1.2 depicts the methodology used to perform the Conceptual database conversion. The physical schema of the legacy database (SPS) is extracted and transformed into a conceptual schema (CS), through a complex

[1]DML standing for Data Management Language

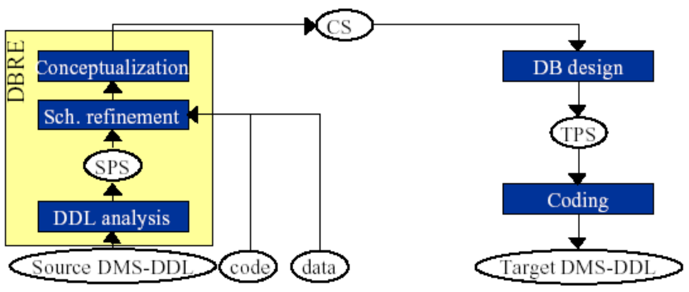


Figure 1.2: *The Conceptual database conversion (D2)*

DBRE process. Then, the physical schema of the target database (TPS) is designed from the CS, through standard database development techniques. The TPS is used to generated the DDL of the target database. The DBRE methodology proposed in [Hen03] consists of three main steps:

- *DDL analysis*: extracting the legacy physical schema (SPS) by parsing the DDL code. The SPS includes all the data structures and constraints *explicitly* declared in the DDL code.

- *schema refinement*: analyzing the SPS and the data management of the legacy programs in order to discover *implicit* constructs and constraints to be added in the schema.

- *data structure conceptualization*: interpreting the refined physical schema into the conceptual schema (CS). Both schemas have the same semantics, but the CS is independent from the data management system.

The Wrapper strategy (P1) The new database obtained through the Conceptual conversion process is encapsulated by a *data wrapper*. A data wrapper is a *data model conversion component* (i.e., a program) that is called by the application programs to carry out operations on the database. Generally, a wrapper simulates the modelling paradigm of a new database, providing a modern interface to a legacy database. In the present context, the wrapper is actually an *inverse* wrapper. It converts all legacy DMS requests from the legacy programs into new DMS requests. Conversely, it captures results from the new DMS, converts them to the legacy format, and delivers them to the calling programs.

Once the wrapper has been written, it has to be interfaced with the legacy applications. This can be done by replacing access statements with wrapper invocations. In the case of COBOL files, each access statement (`READ`, `WRITE`, etc.) can be replaced with a `CALL` statement invoking the wrapper. Figure 1.3 gives an example of such a transformation applying to a COBOL sequential `READ` statement. The `READ` statement

```
READ CUSTOMER NEXT        CALL WR-CUSTOMER
   AT END                    USING "read", CUS, WR-STATUS
      DISPLAY "error"     IF WR-STATUS-AT-END
   NOT AT END                DISPLAY "error"
      DISPLAY CUS-CODE    ELSE
      DISPLAY CUS-DESCR      DISPLAY CUS-CODE
      DISPLAY CUS-HIST.      DISPLAY CUS-DESCR
                             DISPLAY CUS-HIST.

a) before transformation  b) after transformation
```

Figure 1.3: *The Wrapper strategy (P1)*

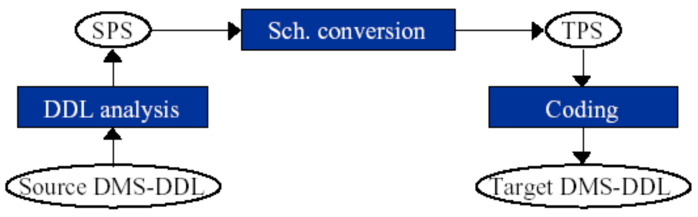


Figure 1.4: *The Physical database conversion (D1)*

is replaced with a `CALL` to a wrapper (`WR-CUSTOMER`), followed by a test of the state of that wrapper (`WR-STATUS`).

1.2.4 The <D1,P2> strategy

Physical conversion (D1) According to the Physical conversion strategy each *explicit* data structure of the legacy database is simply translated into the closest structure in the target DMS. For instance, during the D1 conversion of COBOL files into a SQL database, each record type becomes a table and each top-level field becomes a column.

Figure 1.4 shows the methodology used to perform the Physical database conversion strategy. First, the physical schema of the source database (SPS) is extracted through a DDL analysis process. Then, this schema is converted into its target DMS equivalent TPS, through a straightforward *one-to-one* mapping. The TPS is finally coded into the target DDL.

Statement Rewriting strategy (P2) As seen above, the data structure does not change during the D1 database conversion, but it is just translated according to the new DMS. The DMS statements of the legacy programs can easily be translated into the new DML, such that they directly access the new database instead of the legacy one (P2). For example, a COBOL sequential `READ NEXT` statement would become a `FETCH` SQL query, as shown in Figure 1.5.

```
READ CUSTOMER NEXT        FETCH CURSOR-CUSTOMER
   AT END                    INTO CUS-CODE, CUS-DESCR, CUS-HIST
      DISPLAY "error"     IF SQLCODE NOT = 0
   NOT AT END                DISPLAY "error"
      DISPLAY CUS-CODE    ELSE
      DISPLAY CUS-DESCR      DISPLAY CUS-CODE
      DISPLAY CUS-HIST.      DISPLAY CUS-DESCR
                             DISPLAY CUS-HIST.

a) before transformation  b) after transformation
```

Figure 1.5: *The Statement Rewriting strategy (P2)*

1.3 CASE Support for Database Conversion

1.3.1 DB-MAIN CASE Tool

DB-MAIN is a data-oriented CASE environment [DBM]. Its objective is to support most database engineering processes. It can help developers and analysts in the development, re-engineering, migration and evolution of data-centered applications. DB-MAIN is also a research, development and technology transfer project of the LIBD laboratory of University of Namur. This project started in the early 90's.

Functionalities DB-MAIN offers general functions and components that allow the development of sophisticated processors supporting the data-centered application renovation. Among them:

- A generic model of schema representation based on the *Generic Entity/Relationship* (GER) 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.

- A history processor to record, replay, save or inverse history.

Voyager 2 The Voyager 2 language (V2) is proposed to the user of DB-MAIN to develop new functions which will be seamlessly incorporated in the tool. Voyager 2 is a complete, 4th-generation, semi-procedural language which offers predicative access to the repository of DB-MAIN, the analysis and the generation of external texts, the definition of recursive functions and procedures, etcetera. For instance, a SQL generator for relational compliant GER schema has been written in Voyager 2. Voyager 2 is also used to support wrapper generation.

1.3.2 Database Conversion Support

In the particular context of database conversion, DB-MAIN provides the developer with a toolset for database reverse engineering, mappings definition and wrapper generation [TH01].

Database Reverse Engineering support DB-MAIN offers functions and processors that are specific to database reverse engineering. Among them, let us mention:

- the *data structures extractors*, that identify and parse the data declaration parts of the source code, and create corresponding abstractions of the data physical structure (i.e., the physical schema). The data structures extractors support the first phase of the DBRE process described above: the DDL analysis. Extractors have been built for COBOL, CODASYL, IMS and RPG data structures. Figure 1.6 gives an example of a DB-MAIN physical schema, extracted from a small COBOL program.

- the *data dependency analyzer*, that detects and displays the dependencies between the objects of the program, i.e., variables, constants and records.

- the *program slicer*, that allows the analyst to reduce the search space when he looks for definite information in large programs.

  > *Considering program P, a point p in P (e.g. an instruction) and an object V (a variable or a record), the backward program slice of P with respect to the slicing criterion <p,V> is the set of all statements of P that can contribute to the state of V at point p.* [Hen03]

- a *foreign key discovery assistant*, which proposes some heuristics to find foreign keys in legacy databases. The assistant consists of a set of processors helping the analyst in the discovery of the implicit referential constraints (foreign keys).

The specific DBRE techniques, functions and processors are described in details in [Hai02] and [Hen03].

Mapping Definition Support DB-MAIN can automatically generate and maintain a *history* log of all the transformations that are successively applied to schemas when the analyst carries out any engineering process, e.g., reverse engineering [TH01]. This history is formalized in such a way that it can be analyzed and transformed. In a history, each 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 [HHTH02].

Particularly, the formalized history log can be analyzed in order to derive forward and as well as backward mappings between the source physical schema (SPS) and the target physical schema (TPS).

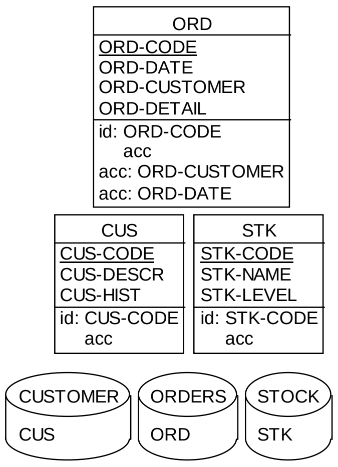


Figure 1.6: *Example of DB-MAIN physical schema*
In this schema, a box represents a physical entity type (record type or table). The first compartment specifies its names, the second one gives its components (fields or attributes) and the third one specifies the keys and other constraints: `id` *stands for primary identifier/key;* `acc` *stands for access key or index ;* `ref` *stands for foreign key. A cylinder represents a data collection (e.g., a file).*

Wrapper Generation Support The wrapper generation in the DB-MAIN environment is performed in two steps, namely *history analysis* (common to all generators) and *wrapper encoding* (specific to a DMS family). The history analyzer parses the schema transformation history and enriches the source/target physical schema with target/source physical correspondences. At the end of this phase, both source and target physical schemas include, for each construct, the rule according which it has been mapped into the other physical schema constructs. In this way, each schema holds all the information required by the wrapper encoder. From the target physical schema (TPS) and the source physical schema (SPS) obtained so far, the wrapper encoder produces the procedural code of the specific wrappers. DB-MAIN wrapper encoders for COBOL files and relational data structures are available. They have been developed in Voyager 2.

1.4 Master's thesis purpose

We shown in Section 1.3 that sophisticated tools already exist to support the conversion of a legacy database (D). This Master's thesis contributes to the automation of the second step of the system migration: the legacy programs conversion (P). In particular, we will try to show that both the *Wrapper* and the *Statement Rewriting* strategies can be fully automated. For each of these two program conversion strategies, we will propose a prototype in the context of the conversion of COBOL files into a SQL database.

1.5 Overview

Chapter 2 presents the COBOL file management system and defines both the program conversion strategies in the particular case of the conversion of COBOL files into a SQL database. In Chapter 3, we present the ASF+SDF Meta-Environment, that we used to automate the COBOL program transformations. Chapter 4 proposes a general methodology for COBOL program transformations, from pre-processing tasks to the pretty-printing of the resulting program. In Chapters 5 and 6 we describe our approach to implement both the program conversion strategies defined in Chapter 2. Both implementations use the ASF+SDF Environment. In Chapter 7, we present a case study by applying our results to a small COBOL application. In Chapter 8 we evaluate our results, our methodology and the tools we used. Finally, Chapter 9 concludes this thesis and presents the future perspectives.

Chapter 2

P1 and P2 Conversions with COBOL

In this chapter, we specify both the *Wrapper* (P1) and the *Statement Rewriting* (P2) program conversion strategies in the case of the conversion of COBOL applications. We assume that (some of) the COBOL files used by the COBOL programs have been migrated to a SQL database. We explain how the COBOL programs can be transformed accordingly, so that they access the new SQL database instead of the migrated files.

This chapter is organized as follows. Section 2.1 introduces the COBOL file system and defines the COBOL data access statements. In Section 2.2, we specify the P1 COBOL conversion strategy, by assuming that a <D2,P1> migration strategy has been chosen. Section 2.3 specifies the P2 COBOL conversion strategy, by assuming that a <D1,P2> migration strategy has been chosen.

2.1 Cobol File Management

2.1.1 The Cobol File

File A COBOL file is an organized collection of related data [Joh86]. A COBOL program can read and write files. Each file is defined in two distinct parts of the program source [Hen03]:

- The `FILE-CONTROL` paragraphs of the `INPUT-OUTPUT` section of the `ENVIRONMENT` division declare the files used, their organization, their access mode, their access keys and their identifiers. Figure 2.1 shows an example of such a paragraph, also called "`SELECT` clause". The indexed file `ORDERS` is declared.

- The `FD` paragraphs of the `FILE` section of the `DATA` division declare the record types with their fields decomposition and the type and the length of the fields. Figure 2.2 shows an example of an `FD` paragraph. The file `ORDERS` is made up records that are decomposed in four fields. The `PIC` clause gives the type and the

```
SELECT ORDERS ASSIGN TO "c:\ORDERS.DAT"
    ORGANIZATION IS INDEXED
    ACCESS MODE IS DYNAMIC
    RECORD KEY IS ORD-CODE
    ALTERNATE RECORD KEY IS ORD-CUSTOMER
      WITH DUPLICATES
    ALTERNATE RECORD KEY IS ORD-DATE
      WITH DUPLICATES.
```

Figure 2.1: *Example of a* `SELECT` *clause*

```
FD ORDERS.
  01 ORD.
   02 ORD-CODE PIC 9(10).
   02 ORD-DATE PIC X(8).
   02 ORD-CUSTOMER PIC X(12).
   02 ORD-DETAIL PIC X(200).
```

Figure 2.2: *Example of an* `FD` *paragraph*

length of the fields. For instance, `PIC 9(10)` means *numeric field of length 10* ; `PIC X(8)` means *alphanumeric field of length 8*.

Record type A COBOL file is made up of records. A record is a collection of related data items treated as a unit. The record type of a file is the set of indivisible data, read or written during the access to the file. The data items that make up a record are called fields. In the case of the file `ORDERS` in Figure 2.2, the record type is `ORD`. Actually, the record type of a COBOL file is the data item declared at the level `01`.

Organization A COBOL file can be organized with three different ways [Bro98]:

- `SEQUENTIAL`: The records are sequenced and are stored and accessed in consecutive order according to this sequence.

- `RELATIVE`: Each record is identified with its order number in the file.

- `INDEXED`: The records may be accessed by the value of a key.

Access Mode The `ACCESS MODE` of a COBOL file is one of the following:

- `SEQUENTIAL`: It is the default. The records are read or written sequentially.

- `RANDOM`: It requires to supply a key to read or write a record.

- `DYNAMIC`: It allows the programmer to read the file with both `SEQUENTIAL` and `RANDOM` accesses. Writing is always `RANDOM`.

Access keys For each `SEQUENTIAL` or `INDEXED` file, the programmer declares a `RECORD KEY`, that is one of the fields identifying each record. For instance, the `RECORD KEY` of the file `ORDERS`, declared in Figure 2.1, is the field `ORD-CODE`.

For each `RELATIVE` file having a `RANDOM` or a `DYNAMIC` access mode, the programmer declares a `RELATIVE KEY`. The `RELATIVE KEY` is a data item apart from the record. Given the current record, the value of its number order in the file is stored in the `RELATIVE KEY` [Cla81].

The *primary key* of a file, also called *identifier*, is the data item used to identify each record in the file. In this thesis, the *primary key* of a COBOL file means:

- either its `RECORD KEY`, for a sequential or an indexed file
- either its `RELATIVE KEY`, for a relative file.

Other data items used as keys are called `ALTERNATE RECORD KEY`s. These keys may provide alternate paths for retrieval of records and are not required to be unique. An `ALTERNATE RECORD KEY` clause can indeed be used `WITH DUPLICATES`, as shown in Figure 2.1.

2.1.2 The Cobol DMS statements

As seen above, for both P1 and P2 conversions, the DMS statements have to be rewritten. The main COBOL file access statements are the following:

- `OPEN`
- `CLOSE`
- `START`
- `READ`
- `WRITE`
- `REWRITE`
- `DELETE`

We will now define the syntax and the effect of these DMS statements more precisely.

OPEN statement

Syntax `OPEN` *open-option file-name*

Effect The option *open-option* affects the opening of *file-name* as follows:

`INPUT`: opens the file and positions it to its start point for **reading**.
`OUTPUT`: creates the file (if necessary) and positions it to its start point for **writing**.
`I-O`: opens the file for both **reading** and **writing**.
`EXTEND`: creates the file (if necessary) and positions it just past the last record in the file for **writing**.

Note that by opening a file, the file buffer is made available for the program. But it does not mean that this buffer is already initialized. Indeed, the `OPEN` statement does not perform any initial reading.

CLOSE statement

Syntax `CLOSE` *file-name*

Effect It simply closes the file *file-name*. Closing files makes them ready for processing by another application.

START statement

Syntax

```
START file-name [KEY IS relational-operator access-key]
      INVALID KEY imperative-statements-1
      [NOT INVALID KEY imperative-statements-2]
END-START
```

with *relational-operator* ϵ {>,>=,= }

Effect The `START` statement positions to a specific record in a relative or indexed file, allowing the programmer to begin reading sequentially from that record. If the `KEY` phrase is omitted, COBOL assumes the primary key of the file is used. The `INVALID KEY` phrase provides statements to execute if a record with the specified key value cannot be found. The optional `NOT INVALID KEY` phrase executes statements if such a record is found. The `INVALID/NOT INVALID` clauses can also be used for the `READ`, `WRITE`, `REWRITE` and `DELETE` statements, in the case of indexed or relative files.

READ statement

There are two kinds of read statement in COBOL, depending on the access mode of the file to read. Moreover, there are two ways of reading a record. First, it can be read `INTO` an identifier. Second, the `INTO` phrase can be omitted and the record is processed directly in the record area; that is in the buffer.

A) Sequential access mode

Syntax

```
READ file-name NEXT [INTO identifier]
    AT END imperative-statements-1
    [NOT AT END imperative-statements-2]
END-READ
```

Effect In the `SEQUENTIAL` access mode, the records are read in the ascending order based on their *reference key* (See below). The `AT END` phrase provides statements to execute when the end of file is encountered. An end of file occurs when an attempt is made to read a record after the last record has been read. The optional `NOT AT END` phrase executes statements if no end of file is encountered, thus, if a record is read.

Reference Key By default, the reference key is the primary key of the file. The reference key of an indexed file can be changed by the execution of a `START` or a random `READ` statement. The specified access key becomes the reference key. Note that a simple static analysis of the program may not allow to figure out which is the reference key of a file at a given point of the program. For instance, the same `READ NEXT` statement can be reached from two distinct `START` statements during the execution of the program. Figure 2.3 gives such an example with the file `ORDERS` declared in Figure 2.1. The reference key used by the sequential read located at line 20 can be either `ORD-CODE` either `ORD-DATE`. It depends on the `PERFORM` statement from where the `READ-ORD-NEXT` paragraph has been called (line 7 or line 16).

B) Random access mode

Syntax

```
READ file-name [INTO identifier] [KEY IS access-key]
    INVALID KEY imperative-statements-1
    [NOT INVALID KEY imperative-statements-2]
END-READ
```

Effect If the access mode is `RANDOM`, the programmer has to supply a key. If the `KEY` phrase is omitted, COBOL assumes the primary key of the file is used. The `INVALID KEY` phrase provides statements to execute if a record with the specified key cannot be found. The optional `NOT INVALID KEY` phrase executes statements if such a record is read.

WRITE statement

There are two ways of writing records. First, a record can be written `FROM` an identifier. Second, the `FROM` phrase can be omitted and the record is moved directly from the buffer.

```
READ-ORD-CODE.
  START ORDERS KEY IS > ORD-CODE
    INVALID KEY
      MOVE 0 TO END-FILE
    NOT INVALID KEY
      MOVE 1 TO END-FILE.
  PERFORM READ-ORD-NEXT
    UNTIL END-FILE = 0.

READ-ORD-DATE.
  START ORDERS KEY IS > ORD-DATE
     INVALID KEY
      MOVE 0 TO END-FILE
    NOT INVALID KEY
      MOVE 1 TO END-FILE.
   PERFORM READ-ORD-NEXT
     UNTIL END-FILE = 0.

READ-ORD-NEXT.
   READ ORDERS NEXT
     AT END
       MOVE 0 TO END-FILE
     NOT AT END
       ...
```

Figure 2.3: *The* `READ NEXT` *reference key*

A) Sequential access mode

Syntax `WRITE` *record-name* [`FROM` *identifier*]

Effect In the `SEQUENTIAL` access mode, records are written sequentially. The *record-name* is the name of the level 01 entry, described in the `FILE` section of the `DATA` division.

B) Random access mode

Syntax

WRITE *record-name* [FROM *identifier*]
    INVALID KEY *imperative-statements-1*
    [NOT INVALID KEY *imperative-statements-2*]
END-WRITE

Effect In the `RANDOM` access mode, `INVALID KEY` executes statements if a record already exists with the same record key value. The optional `NOT INVALID KEY` phrase executes statements if the record is written.

REWRITE statement

A) Sequential access mode

Syntax `REWRITE` *record-name* [`FROM` *identifier*]

Effect The `REWRITE` statement locates a specified record in the file and replaces it with the content current record value (or the content of *identifier*). For a `SEQUENTIAL` access, a record must be read before it can be rewritten.

B) Random access mode

Syntax

```
REWRITE record-name [FROM identifier]
    INVALID KEY imperative-statements-1
    [NOT INVALID KEY imperative-statements-2]
END-WRITE
```

Effect When access is `RANDOM` or `DYNAMIC`, a value has to be moved to the record key before rewriting the record. The `INVALID KEY` phrase executes statements if the file does not contain any record with the same record key value. The optional `NOT INVALID KEY` phrase executes statements if such a record is rewritten.

DELETE statement

The `DELETE` statement deletes records.

A) Sequential access mode

Syntax `DELETE` *file-name* [`RECORD`]

Effect When the file access is `SEQUENTIAL`, the record must be read successfully before being deleted.

B) Random access mode

Syntax

```
DELETE file-name
    INVALID KEY imperative-statements-1
    [NOT INVALID KEY imperative-statements-2]
END-DELETE
```

Effect For the `RANDOM` or `DYNAMIC` access modes, a value has first to be moved to the record key, to indicate the record to delete. If the record key is invalid, the `INVALID KEY` phrase executes the given statements.

2.2 The P1 conversion in the <D2,P1> Strategy

As seen above, the P1 program conversion strategy assumes that a wrapper encapsulates the new database obtained through the database conversion step. The P1 conversion applied to COBOL programs consists of replacing the COBOL DMS statements (accessing the files that have been migrated) with a wrapper invocation. Furthermore, the P1 conversion must reorganize other portions of the Cobol program, e.g., the declarations of the migrated files.

A COBOL program contains 4 divisions:

- `IDENTIFICATION DIVISION`: containing comments identifying the program, its author, and the date it was written
- `ENVIRONMENT DIVISION`: naming the source and object computer and describing each file used by the program
- `DATA DIVISION`: describing all data items
- `PROCEDURE DIVISION`: containing the executable program statements

The P1 conversion transforms the last three divisions. We will now describe more precisely what these divisions contain, and how they can be modified according to the P1 strategy.

2.2.1 Environment Division

The `ENVIRONMENT` division consists of two optional sections: the `CONFIGURATION` section and the `INPUT-OUTPUT` section. In the latter, we can find the file definitions. Each file must be named in a separate `SELECT` clause. The `SELECT` clause associates the external file name with the name used to reference it in the program. This association is made in the `ASSIGN` clause. The `SELECT` clause may also provide information on:

- the organization of the file
- the access mode of the file
- the primary key of the file
- the `ALTERNATE`'s keys of the file

The Figure 2.4 shows an example of an `ENVIRONMENT` division.

Once the program has been transformed, it does not access the migrated files anymore. Thus, their `SELECT` clauses can be removed from the `ENVIRONMENT` division.

```
ENVIRONMENT DIVISION.
  INPUT-OUTPUT SECTION.
  FILE-CONTROL.
    SELECT CUSTOMER ASSIGN TO "c:\CUSTOMER.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS CUS-CODE.
    SELECT ORDERS ASSIGN TO "c:\ORDERS.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS ORD-CODE
      ALTERNATE RECORD KEY IS ORD-CUSTOMER
      WITH DUPLICATES
      ALTERNATE RECORD KEY IS ORD-DATE
      WITH DUPLICATES.
    SELECT STOCK ASSIGN TO "c:\STOCK.DAT"
      ORGANIZATION IS INDEXED
      ACCESS MODE IS DYNAMIC
      RECORD KEY IS STK-CODE.
```

Figure 2.4: *Example of* ENVIRONMENT *division*

2.2.2 Data Division

The DATA division consists of the following sections:

- FILE SECTION: to specify each file and its records
- WORKING-STORAGE SECTION: to describe all data items and Working-Storage records
- LOCAL-STORAGE SECTION : to describe local storage used when the subprogram is called
- LINKAGE SECTION: to describe parameters in a called subprogram

Two DATA division transformations are required. First, the record definitions of the migrated files have to be moved from the FILE section to the WORKING-STORAGE section. In this way, the old file buffers are declared as any other variable of the program. Second, several new variables have to be declared in the WORKING-STORAGE section. Figure 2.5 gives an example of a DATA division transformation. In this example, we want to migrate the files CUSTOMER, ORDERS and STOCK. The three record definitions (CUS, ORD and STK), previously located in the FILE section (lines [4-7], [10-14] and [17-20] of Figure 2.5a), are moved to the WORKING-STORAGE section (lines [3-6], [8-12] and [14-17] of Figure 2.5b). In this way, the three old file buffers are now considered as any other variable of the program. The FD paragraphs of the files CUSTOMER, ORDERS and STOCK, are removed from the FILE section, making this section empty. So the optional FILE section can be removed from the DATA division.

```
DATA DIVISION.
FILE SECTION.
FD CUSTOMER.
01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(110).
  02 CUS-HIST PIC X(1000).

FD ORDERS.
01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-DATE PIC X(8).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).

FD STOCK.
01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

WORKING-STORAGE SECTION.
01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDR PIC X(40).
  02 COMPANY PIC X(30).
  02 FUNCT PIC X(10).
  02 REC-DATE PIC X(10).
        ...
```

a) before transformation

```
DATA DIVISION.
WORKING-STORAGE SECTION.
01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(110).
  02 CUS-HIST PIC X(1000).

01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-DATE PIC X(8).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).

01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDR PIC X(40).
  02 COMPANY PIC X(30).
  02 FUNCT PIC X(10).
  02 REC-DATE PIC X(10).
        ...
```

b) after transformation

Figure 2.5: *Example of* `DATA` *division transformation*

2.2.3 Procedure Division

Wrapper invocation

Remember that P1 has to replace each access statement with a call to a wrapper. In COBOL, a wrapper (that is an external program) can be invoked using the `CALL` statement. In this project, there is a wrapper for each record type, and thus for each file used in the program. So the wrapper to call depends on the file to access. The COBOL wrapper invocation has the following general form:

```
CALL wrapper-name USING operation
                        record-name
                        option
                        wrapper-status
```

such that:
- *operation* specifies the access operation to perform
- *record-name* is the record name of the COBOL file
- *option* specifies an optional parameter.
- *wrapper-status* indicates if the operation has been successfully performed.

The first, third and fourth arguments must be declared as new variables in the `WORKING-STORAGE` section.

Wrapper specifications

The wrapper access the SQL database and simulates the DMS statement *operation* used with the corresponding *option*. For instance, let us consider the following call:

```
CALL WR-CUSTOMER USING "read"
                       CUS
                       "KEY IS CUS-CODE"
                       WR-STATUS
```

The wrapper of the file `CUSTOMER` will preserve the semantics of the COBOL random `READ`. It will move to the buffer (`CUS`) the first logical record having the same key value as the current value of `CUS-CODE`. If a logical record with the specified key value cannot be found, the `INVALID KEY` exception will be indicated into `WR-STATUS`.

2.3 The P2 conversion in the <D1,P2> Strategy

The P2 transformations of the `ENVIRONMENT` and `DATA` divisions are the same as in P1 strategy. They consist of reorganizing the files and data declarations in these divisions.

```
FD ORDERS.
01 ORD.
02 ORD-CODE PIC 9(10).
02 ORD-DATE PIC X(8).
02 ORD-CUSTOMER PIC X(12).
02 ORD-DETAIL PIC X(200).
```

a) COBOL file declaration

```
create table ORD(
ORD_CODE numeric(10) not null,
ORD_DATE char(8) not null,
ORD_CUSTOMER char(12) not null,
ORD_DETAIL char(200) not null,
primary key (ORD_CODE));

create index ORD_CUSTOMER
on ORD(ORD_CUSTOMER);

create index ORD_DATE
on ORD(ORD_DATE);
```

b) SQL table declaration

Figure 2.6: *D1 translation of a COBOL file into a SQL table*

2.3.1 Procedure Division

In the P2 conversion strategy, we rewrite each access statement of a legacy program, according to the new Data Management System. In our project, the latter is `SQL`. Basically, we have to replace the COBOL access statements described previously with `SQL` statements. Each COBOL record type is now a `SQL` table, and that each top level field of this record, is now a column of this table. Figure 2.6 shows an example of a D1 translation. The COBOL file `ORDERS`, declared in Figures 2.4 and 2.5, becomes the `SQL` table `ORD`. We can see that D1 does not change the data structure, and preserves the access keys declaration. The primary key of the file `ORDERS` (`ORD-CODE`) is translated into a SQL column declared as primary key of the table `ORD`. The alternate keys `ORD-CUSTOMER` and `ORD-DATE` are translated into two SQL indexes on the table `ORD`. The SQL columns are required to be `not null` since the COBOL fields they translate must be initialized in each record of the file.

SQL cursors We will now describe how we can transform the COBOL access statements so that the program accesses the SQL tables instead of the migrated files. Actually, there is a difference between COBOL and SQL data access. In COBOL, the `READ` returns a **single record**. In SQL, the `SELECT` clause returns a **set of rows**. This is the reason why we have to use SQL cursors. A SQL cursor allows to select a set of rows from a table, that can be fetched sequentially into COBOL variables. Here is an example of a cursor declaration:

```
DECLARE GE_ORD_DATE CURSOR FOR
 SELECT ORD_CODE, ORD_DATE, ORD_CUSTOMER, ORD_DETAIL
 FROM ORD
 WHERE ORD_DATE => :ORD-DATE
 ORDER BY ORD_DATE
```

In this example, the file `ORDERS` (declared in Figures 2.4 and 2.5) has been converted into the table `ORD`. Each column of this table is the translation of a field of the record type `ORD`. The declared cursor selects the rows of the table `ORD` where the value of the column `ORD_DATE` is greater or equal to the value of the field `ORD-DATE`.

Before fetching a cursor, the latter has to be open. When opening a cursor, it is evaluated in accordance with the current state of both the SQL database and the COBOL variables. We can open the SQL cursor we declared above with the following statement:

```
OPEN GE_ORD_DATE
```

Then we can fetch sequentially a row of this cursor, into the corresponding COBOL variables:

```
FETCH GE_ORD_DATE
  INTO :ORD-CODE,:ORD-DATE,:ORD-CUSTOMER,:ORD-DETAIL
```

Cursor-based loop The change of paradigm when moving COBOL 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 IS` and `READ KEY IS`), then to go on in this sequence through `READ NEXT` statements. The most obvious SQL translation is through a cursor-based loop. However, we shown in Figure 2.3 that the `READ NEXT` statements can be scattered throughout the program, so that the identification of initial `START` statement, specifying the reference key, is complex. The technique illustrated in Figure 2.7 solves this problem. A cursor is declared for each kind of access key usage (`=`, `>`, `>=`) in the program. For instance, the table `ORD` gives six cursors, resulting of the combination of two access keys and three keys usages. A `START` can be translated through the corresponding cursor opening. The following `READ NEXT` is to be replaced with a `FETCH` of *that* cursor. The variable `LAST-CURSOR-ORD` specifies the name of the currently open cursor on the table `ORD`. By testing the value of this variable, we can figure out which cursor has to be fetched, with respect to the current reference key.

Note that Figure 2.7 only gives an intuitive idea of the P2 transformation. For instance, we omit here the simulation of the `INVALID KEY` and `AT END` exceptions. We refer to Chapter 6, *P2 Implementation*, for more details.

Here are the other COBOL-SQL translation rules:

- A random `READ KEY IS` can be regarded as a `READ NEXT` that immediately follows a `START KEY IS EQUAL`. So a random `READ` can be replaced with a cursor opening and a `FETCH` of this cursor.
- A COBOL `WRITE` can be simulated using a SQL `INSERT`
- A COBOL `REWRITE` can be replaced with a SQL `UPDATE`.
- A COBOL `DELETE` can be translated into a SQL `DELETE`.
- The SQL translation of the COBOL `OPEN` consists of opening the cursor that selects all the rows of the table in the ascending order based on the primary key.
- There is no SQL correspondence for the COBOL `CLOSE`. In the <D1,P2> strategy, the `CLOSE` statements can be removed or replaced with the `CONTINUE` statement.

```
                                      EXEC SQL
                                        DECLARE CURSOR GE_ORD_DATE FOR
                                        SELECT ORD_CODE,
                                               ORD_DATE,
                                               ORD_CUSTOMER,
                                               ORD_DETAIL
                                        FROM ORD
                                        WHERE ORD_DATE => :ORD-DATE
                                        ORDER BY ORD_DATE
                                      END-EXEC
                                       ...

START ORDERS              =>          EXEC SQL
  KEY IS => ORD-DATE                    OPEN GE_ORD_DATE
                                      END-EXEC
                                      MOVE "GE_ORD_DATE" TO LAST-CURSOR-ORD

  ...                                  ...

READ ORDERS NEXT          =>        IF LAST-CURSOR-ORD = "GE_ORD_DATE"
                                     THEN
                                      EXEC SQL
                                        FETCH GE_ORD_DATE
                                        INTO :ORD-CODE,
                                             :ORD-DATE,
                                             :ORD-CUSTOMER,
                                             :ORD-DETAIL
                                      END-EXEC
                                    ELSE IF LAST-CURSOR-ORD = "GE_ORD_CUSTOMER"
                                     THEN
                                      EXEC SQL
                                        FETCH GE_ORD_CUSTOMER
                                        INTO :ORD-CODE,
                                             :ORD-DATE,
                                             :ORD-CUSTOMER,
                                             :ORD-DETAIL
                                      END-EXEC
                                    ELSE IF...
                                    END-IF
```

Figure 2.7: *Transformation of a* START/READ NEXT *sequence*

Remarks

- For several reasons, that we will explain later, we decided to generate a separate COBOL section containing all the new SQL code. Each paragraph of this additional section will translate into SQL a kind of COBOL DMS statement. In this way, an access statement can be replaced with a `PERFORM` statement, executing the corresponding paragraph. Instead of calling an external program (a P1 wrapper) we call an internal procedure (a P2 paragraph) to access the new database in the new DML. For instance, we replace the statement "`DELETE ORDERS`" with the statement "`PERFORM P2-DELETE-ORDERS`". The paragraph `P2-DELETE-ORDERS` is a new generated paragraph containing the corresponding SQL `DELETE`.

- As shown in Figure 2.7, the SQL calls can be embedded in the COBOL source code. This feature is provided by a lot of COBOL dialects. The standard syntax of an embedded SQL piece of code is the following:

```
EXEC SQL
  write here the SQL code
END-EXEC
```

2.4 Transformation rules

In appendix A, we give a summary of transformation rules, used to replace the COBOL DMS statements according to both <D2,P1> and <D1,P2> strategies. Remember that

- In the <D2,P1> strategy, we locally replace the COBOL DMS statements with a `CALL` statement to an inverse wrapper.

- In the <D1,P2> strategy, we locally replace the COBOL DMS statements with a `PERFORM` statement executing a generated paragraph.

In both cases, these rules allow the program to access the SQL tables instead of the migrated COBOL files. These rules are not exhaustive, we give *one* rule for each kind of COBOL access statement (`OPEN`, `CLOSE`, `START`, sequential `READ`, random `READ`, etc.). This is a high level representation of the transformations rules. For example, let us consider a COBOL random `READ` statement.

Syntax

```
READ file-name KEY IS access-key
    INVALID KEY statements-1
    NOT INVALID KEY statements-2
END-READ
```

<D2,P1>

```
CALL wrapper-name
  USING "read "
        record-name
        "KEY IS access-key "
        wrapper-status
IF  wrapper-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-READ-file-name-KEY-IS-access-key
IF  p2-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

In Chapter 6, we will describe precisely what the generated P2 paragraphs contain.

Chapter 3

A Programs Transformation Tool: the ASF+SDF Meta-Environment

In this Chapter, we present the ASF+SDF Meta-Environment, that we used in order to automate both P1 and P2 COBOL transformations. The ASF+SDF MetaEnvironment is developed by the *SEN1* research group of the CWI[1]. The *SEN1* group performs research in the area of software reverse engineering, software reengineering and software renovation. In particular, one of the *SEN1* main research topics is *source code transformation*.

3.1 Overview

The ASF+SDF Meta-Environment is an interactive development environment for the automatic generation of interactive systems for manipulating programs, specifications, or other texts written in a formal language [4]. It is a complete system that can be used for different purposes:

- To describe the syntax and semantics of programming languages

- To describe analysis and transformation of programs written in such programming languages

ASF+SDF is a specification language for defining both the syntax and the semantics of programming languages. It is the result of combining two formalisms: ASF (Algebraic Specification Formalism) and SDF (Syntax Definition Formalism).

[1]The CWI is the National Research Institute for Mathematics and Computer Science in the Netherlands.

```
module migration
  %% here starts the SDF syntax definition
    exports
      sorts Word

      lexical syntax
        [a-z]+      ->  Word
        [\ \t \n]    -> LAYOUT

      context-free syntax
        cobol-to-sql(Word)  -> Word

    variables
      "myword"     -> Word

  %% here starts the ASF equations definition
    equations
      [equ1] cobol-to-sql(myword) = sql
          when myword = cobol

      [default-1] cobol-to-sql(myword) = myword
```

Figure 3.1: *Example of an ASF+SDF module*

3.2 ASF+SDF Concepts

3.2.1 Modules

An ASF+SDF specification consists of a collection of module declarations. Each module may define syntax rules (SDF syntax) as well as semantics rules (ASF equations). Conceptually, a module is a single logical unit. But, for technical reasons, the syntax sections and the equations section are stored in physically separate files. For each module M in a specification two files exist: 'M.sdf' containing the syntax sections of M and 'M.asf' containing the equations section of M.

Example The Figure 3.1 shows a simple example of an ASF+SDF module. The module `migration` exports one *sort*, `Word`. In the *lexical syntax*, a `Word` is defined as a non-empty sequence of alphabetic characters. A `LAYOUT` can be either a space (`\ `), a horizontal tabulation (`\t`) or a newline character (`\n`).

In the *context-free syntax*, we define the *function* `cobol-to-sql` which takes a *term* of sort `Word` as an argument and returns a term of sort `Word` as a result.

We define the *variable* `myword`, which is of sort `Word`. This variable will be used in the *equations* section.

In the equation `[equ1]`, we learn that the function `cobol-to-sql` replaces a word with `"sql"` **if** this word is `"cobol"`. This equation is called a *conditional equation.*

```
module top-module
  imports migration
   exports
     sorts Sentence

   context-free syntax
       Word+ "." -> Sentence
```

Figure 3.2: *Example of a module that imports another module*

The equation `default-1` is a special equation. It will be applied only if no other equation can successfully be applied. It expresses that in all other cases, words remain unchanged.

We will now define precisely the different concepts of an ASF+SDF module. Because of the limited scope of this master's thesis, we will only explain what is necessary to understand the chapters that follow. See [vK03] for more details.

3.2.2 SDF comment convention

The comment convention within an SDF specification is that every character between `"%%"` and the end of line is comment as well as every character between two `"%"` including the newline character. Examples of comments are given in Figure 3.1.

3.2.3 Imports

The entities declared in a module may be visible or invisible to other modules. A module can use another module from the specification by importing it. As a result, all visible names of the imported module become available in the importing module. For instance, the module `migration` in Figure 3.1 is imported by the module `top-module` in Figure 3.2. It means that the entities defined in the `exports` section of the module `migration` are available in the module `top-module`. In particular, the sort `Word` can be used in the module `top-module`.

3.2.4 Lexical syntax

The lexical syntax describes the low level structure of text by means of lexical tokens. A lexical token consists of a sort name and the actual text of the token. The sort name is used to distinguish classes of tokens (e.g., identifiers, numbers, etc.).

The lexical syntax also defines which substrings of the text are layout symbols or comments and are to be skipped during syntax analysis. The sort name `LAYOUT` is typically used for defining layout and comment conventions. The Figure 3.1 shows an example of a `LAYOUT` definition. The module `Layout` in Figure 3.3 defines the SDF comment convention.

```
module Layout
 %% In this module we define the
 %% comment convention for SDF.

  exports
    lexical syntax
       "%%" ~[\n]* [\n] -> LAYOUT
       "%" ~[\n%]+ "%" -> LAYOUT
```

Figure 3.3: *The SDF comment*

3.2.5 Context-free syntax

The context-free syntax describes the concrete and abstract syntactic structure of sentences in a language. A context-free syntax contains a set of declarations for context-free functions, each consisting of zero or more symbols followed by `"->"` and a result symbol. The function `cobol-to-sql` in Figure 3.1 is an example of such a function.

3.2.6 Conditional Equations

In ASF+SDF, a conditional equation can be written in three syntactically different ways, which are all semantically equivalent. Here is one of them:

`[TagId]` L = R `when` C_1, C_2, . . . , C_n

where

- `TagId` is a sequence of letters, digits, and/or minus signs.
- L (left-hand side) and R (right-hand side) are of the same sort.
- $C_1, C_2, ..., C_n$ are conditions. Each condition consists of a left-hand side and a right-hand side, and at most one of theses can introduce new variables.
- L is not a single variable.
- The variables that occur in R also occur in L or in one of the conditions.

The left-hand side and the right-hand side of the equation are terms in *concrete* syntax, i.e., user defined syntax, augmented with sorted variables. The conditions can be *positive* or *negative*. They are evaluated one after the other. Only if the evaluation of a condition C_i is successful the rest of the conditions $C_{i+1}, ..., C_n$ is evaluated. A conditional equation is applied if:

1. the left-hand side matches and
2. all the conditions are successfully evaluated

3.2.7 Lexical Constructor Function

For each sort L that appears as result sort in the lexical syntax a *lexical constructor function* of the form `l "(" CHAR+ ")"` -> L is automatically added to the context-free syntax of the specification. Here, `l` is the name of sort L written in lower-case letters. In this way, you can get access to the characters of lexical tokens. For instance, let see in Figure 3.1. The sort `Word` is defined in the lexical syntax of the module `migration`. So the following lexical constructor function is automatically added to the context free syntax:

```
word "(" CHAR+ ")" -> Word
```

In this way, we can get access to the characters of a `Word`. First, we define the variable `char+` as follows:

```
"char+"[0-9]* -> CHAR+
```

Thanks to this definition, we can use the variables `char+1`, `char+2`, `char+3`,... in the equations of the same module. Here we will use the variable `char+1`.

```
[adds1] add-s(myword) = word(char+1 "s")
          when myword = word(char+1)
```

In the equation above, the function `add-s` adds the character `"s"` at the end of a `Word`. By using the lexical function `word` in the condition, the variable `char+1` contains the sequence of the characters of the initial word `myword`. The resulting word, constructed by the lexical constructor function, is simply obtained by adding the character `"s"` after the variable `char+1`. For instance, if the initial word is `"file"`, the variable `char+1` will contain the following sequence of character: `"f" "i" "l" "e"`. Thus the word constructed by `word(char+1 "s")`, is actually `word("f" "i" "l" "e" "s")`, and will be rewritten as the word `"files"`.

3.2.8 Term Rewriting

Term rewriting is the exhaustive application of a set of rewrite rules to a term until no rule can be applied anywhere in the term. This process is also called normalization.

In the ASF+SDF Meta-Environment, equations can be seen as (conditional) rewrite rules. They are used to reduce some initial *closed term*[2] to a *normal form*[3] by repeatedly applying rules from the specification.

For instance, the closed term `"cobol-to-sql(cobol)"` can be reduced to the normal form `"sql"`, by applying the rule defined in the equation `[equ1]` in Figure 3.1.

A term is always reduced in the context of a certain module, say M. The rewrite rules that may be used for the reduction of the term are the rules declared in M itself and in the modules that are imported by M[4] [vK03].

Both interpretation and compilation of the rewrite rules are supported in the ASF+SDF Meta-Environment. In our project, we used the interpretation. The reasons are discussed in Chapter 4.

[2]i.e. a term not containing variables
[3]i.e. a term that is not reducible any further
[4]directly or indirectly

3.2.9 Traversal Functions

Program analysis and program transformation usually take the syntax tree of a program as starting point. The question is: how can the traversal of this tree be expressed ?

The kind of nodes that may appear in a program's syntax tree are determined by the grammar of its programming language. Generally, each rule in the grammar corresponds to a node category in the syntax tree. Most of the programming languages are described by grammars which can contain several hundred of grammar rules. For instance, the COBOL grammar we used contains about 900 syntax productions. In these conditions, a naive recursive traversal function should consider many node categories and the size of its definition will grow accordingly.

This could become even worse since the traversal function usually do some real work (extracting information, make some modifications) only for very few node categories. For instance, in our project we mainly focus on the data access statements of the COBOL program.

Traversal functions in ASF+SDF solve this problem. In addition, they allow passing additional parameters to the transformation and returning additional results.

There are three kinds of traversal functions [vKV03]: the *Accumulators*, the *Transformers* and *Accumulating transformers*. We will now define each of them, and explain why they can be useful in our project.

Accumulator

An accumulator function is a mapping that traverses its first argument, and keeps the accumulated value in its second argument.

The general definition of an accumulator is the following:

$$f(S_1, S_2, ..., S_n) \rightarrow S_2\{traversal(accu)\}$$

An accumulator function does not change the first argument, only its accumulate argument. The accumulated value is an information we want to extract from a tree (i.e. the first argument).

In our COBOL transformations, we need to know the record name corresponding to each file, and the primary key declared for each file as well. By using an accumulator, we can retrieve this information.

Transformer

A transformer function is a transformation that traverses and transforms its first argument.

The general definition of a transformer is the following:

$$f(S_1, S_2, ..., S_n) \rightarrow S_1\{traversal(trafo)\}$$

This is a type-safe transformation, i.e., it always returns the same sort.

In our COBOL transformations, we have to replace each access statement with a call to a wrapper. For this purpose, we need additional data:

1. the table <file : record-type>.
2. the table <file : primary-key>.

These additional arguments can be used during the traversal of a COBOL program, but they cannot be modified.

Accumulating Transformer

An accumulating transformer is an accumulator and a transformer: it accumulates information in its second argument while traversing its first argument.

The general definition of an accumulating transformer is the following:

$$f(S_1, S_2, ..., S_n) \to S_1 \sharp S_2 \{traversal(accu, trafo)\}$$

The return value of an accumulating transformer is the tuple consisting of the traversed (and transformed) first argument and the accumulated value.

3.3 ASF+SDF Tools

The ASF+SDF Meta-Environment provides components that can be used independently of the interactive environment. We will mention some of them in this section, focusing on ones we used in our project.

3.3.1 Parse Table Generation

The *parse table generator* takes an SDF syntax definition as input and generates a *parse table* used for parsing a term. Using the command `pt-dump`, we can construct the parse table corresponding to a top-level module, say M.

```
pt-dump -m M -o M.tbl
```

The resulting parse table `M.tbl` contains a representation of the SDF syntax of the module M.

3.3.2 Parsing

Once the parse table for module M has been constructed, terms can be parsed by using the command `sglr`.

```
sglr -p M.tbl -i term.txt -o term.tree
```

If the term `term.txt` has been successfully parsed, the resulting syntax tree of the term `term.tree` is constructed. If the input term contains a syntax error, the output of `sglr` is an error message.

The `sglr` parser is a "Scannerless Generalized LR parser"[vK03].

- *Scannerless* means that no scanner is used to tokenize the input stream. Lexical analysis (tokenization) and context-free analysis (parsing) are uniformly handled by a single tool.

- *Generalized* means that the parser finds all possible derivations for a certain input string. This implies that the parsing process may be indeterminate, i.e., can have several equally correct results. In other words, the `sglr` parser does not complain about conflicts in the parse table. In this way arbitrary context-free grammars can be parsed.

- An LR parser "reads its input from Left to right and produces a Rightmost derivation"[Wik].

3.3.3 Obtaining equations

In order to apply the rewrite rules defined for the module M, we need the *equations table* of the module M. Using the command `eqs-dump` we can construct this equations table.

```
eqs-dump -m M -o M.eqs
```

The resulting equations table `M.eqs` contains a representation of the ASF equations of the module M and of the modules directly or indirectly imported by M.

3.3.4 Term Rewriting

Given the syntax tree of the term `term.tree` and the equations table `M.eqs`, we can reduce the terms using the command `asfe`.

```
asfe -i term.tree -e M.eqs -o reduct.tree
```

3.3.5 Term Unparsing

Using the command `unparsePT` the resulting normalized term `reduct.tree` can be converted to a textual representation.

```
unparsePT -i reduct.tree -o reduct.txt
```

3.4 ASF+SDF Library

The ASF+SDF Meta-Environment provides a library of ASF+SDF specifications. This library contains examples, reusable specifications such as data-structures and also some grammars for programming-languages.

Chapter 4

Cobol Program Transformations

Before transforming a COBOL program, we need to parse it. We saw previously that we can use the ASF+SDF Meta-Environment to achieve this task. So we need an SDF syntax definition of the COBOL grammar. Such a definition already exists: the `IBM-VSII` COBOL Grammar, discussed in Section 4.1. In Section 4.2, we discuss a related tool, allowing grammar adaptation and deployment : the Grammar Deployment Kit.

In our project, we had to apply some pre-processors to COBOL programs in order to make them parsable with the `IBM-VSII` grammar. We discuss these pre-processors in Section 4.3.

Once the COBOL programs have been parsed and transformed, they have to be printed correctly. This pretty-printing task, consisting of indenting correctly the output COBOL source program, is discussed in Section 4.4.

At the end of this chapter (Section 4.5) we summarize our approach for COBOL Program Transformations by presenting a general methodology.

4.1 IBM-VSII Cobol Grammar

The IBM-VSII COBOL grammar has been derived from IBM's *VS COBOL II Reference Summary* in a semi automatic, formal process [LV01]. The authors of this grammar are Ralf Lämmel and Chris Verhoef, from the Free University of Amsterdam. The IBM-VSII grammar was expressed using a variant of `EBNF` (Extended Backus Naur Form), the so-called `LLL` grammar format, that is the formalism of the Grammar Deployment Kit. The process started with the master's thesis of an undergraduate student. It took him four months to specify about 1100 production rules. It seems to be the largest specification of `COBOL85`[1] even written [vSV97]. Then, almost all the context-free grammar rules of the IBM syntax were translated into SDF rules. This translation was a very complex process, especially because the grammar has to be disambiguated and better restructured. The resulting COBOL IBM-VSII SDF grammar consists of 16 modules, 552 sorts and 906 productions. A large number of test programs were parsed by the generated parser, which validated the correctness and the completeness of the resulting context-free grammar.

The authors of the grammar emphasize that their approach "*is not just a reduction of an existing `COBOL85` grammar but it uses it as a reference. If constructs that are not present in the standard are located in the code we add those to our grammar*" [vSV97]. They also used

[1]`COBOL85` is the third version of COBOL, released in 1985.

this approach to define SQL syntax, in order to allow embedded calls occurring in programs written in COBOL dialects.

4.2 The Grammar Deployment Kit

«*Grammar deployment is the process of turning a given grammar specification into a working parser.*» [Kor03] The Grammar Deployment Kit (GDK) is a free software that provides tool support for grammar adaptation and parser generation [KLV02].

Note that we did not use the Grammar Deployment Kit during our project. The available IBM-VSII COBOL syntax was suitable to parse the programs we had to transform. We present GDK as an inspiring idea for future developments.

The Grammar Deployment Kit provides four important components:

- `LLL`: an `EBNF`-like grammar format.
- `LLLimport`: a tool to import grammars from another format to the `LLL` format (SDF, YACC, etc.).
- `FST`: a tool to transform grammars. It provides 16 distinct operators.
- `LLLexport`: a tool to export grammars, from the `LLL` format to another format. In particular, `LLLexport` can be used to export a `LLL`-grammar to SDF.

We will focus on the `FST` tool. Thanks to the latter, a `LLL` initial grammar which is still ambiguous or incomplete, can be adapted and completed. This was the case of the first version of COBOL IBM-VSII. Many problems in the IBM document have been detected and fixed in a traceable way. For instance, ambiguities occurred because of the `DELETE` statement, defined as follows:

```
delete-statement :
    "DELETE" file-name "RECORD"?
    invalid-key-statement-list?
    not-invalid-key-statement-list?
    "END-DELETE"? ;
```

With this syntax rule, we do not know how to parse the following piece of code.

```
DELETE CUSTOMER
    INVALID KEY DISPLAY "ERROR"
ACCEPT REC-DATE
MOVE DESCRIPTION TO CUS-DESCR
PERFORM INIT-HIST.
```

Actually, an ambiguity is possible when the `DELETE` statement is not explicitly closed with an `END-DELETE`. The piece of code above can be regarded as a single `DELETE` statement as well as many statements. The GDK script, shown in Figure 4.1, solves this problem, by restructuring the syntax definition of the `DELETE` statement. After the grammar adaptation performed by the script, we know that an unclosed `DELETE` can not be followed by other statements in the same sequence of statements (i.e., in the same `statement-list`). Thus, there is now only one way to parse the portion of code above: as an unclosed `DELETE` statement.

```
%introduce
delete-statement-simple:
  "DELETE" file-name "RECORD"?;

%introduce
invalid-key-phrases:
  invalid-key-statement-list
  not-invalid-key-statement-list?
| not-invalid-key-statement-list;

%introduce
delete-statement-non-closed:
  delete-statement-simple
  invalid-key-phrases;

%redefine
delete-statement :
  "DELETE" file-name "RECORD"?
  invalid-key-statement-list?
  not-invalid-key-statement-list?
  "END-DELETE"? ;
%to
delete-statement:
  delete-statement-simple
| delete-statement-simple "END-DELETE"
| delete-statement-non-closed "END-DELETE";

%include
statement-non-closed:
  delete-statement-non-closed;

%redefine
statement-list:
  statement* ;
%to
statement-list:
  statement*
  statement-non-closed?;
```

Figure 4.1: *The* `DELETE` *FST script*

The script in Figure 4.1 uses three FST operators. The `%introduce` operator allows to introduce a new rule in the grammar. The `%redefine` operator gives a new definition for an existing rule. The `%include` operator adds the disjunction of the rule to the existing one [Kor03].

In the scope of this work, the Grammar Deployment Kit can be regarded as an ad-hoc tool to adapt grammars in a traceable manner. Imagine that we need to extend our ASF+SDF specifications in order to deal with a specific COBOL dialect. For the syntax part adaptation, we would recommend to use GDK. First, we would *import* the SDF grammar to the LLL-format, using `LLLimport`. Second, we would *transform* the grammar using the different operators of `FST`. Last, we would *export* the resulting LLL-grammar to SDF, using `LLLexport`.

4.3 Pre-processing

The COBOL IBM-VSII grammar has to be seen as a specification rather than a realistic parser description. Several issues, usually handled by a pre-processor, are not covered

```
* I like writing comments  %%* I like writing comments
* in my program            %%* in my program
/                          %%/
DISPLAY "NEW PAGE".        DISPLAY "NEW PAGE".
```

a) before pre-processing   b) after pre-processing

Figure 4.2: *Comments Pre-processing phase*

by the formal definition. An example is the case of the COBOL comments.

A COBOL comment is every character between an asterisk (`*`), written in column 7 of a line, and the end of this line. Typically, comments are used to enter explanations about a portion of the program. Since comments may occur on every location in the code, it would be a nightmare to incorporate them in the SDF context-free grammar. It is the same problem with the slash (`/`) frequently written in the column 7, causing a portion of the program to be listed on a new page on the printer.

Thus, in order to be able to parse programs that contain comments and slashes, a pre-processing phase is required. The idea of the pre-processing phase is quite simple. We replace all the COBOL comments and slashes in the program with SDF comments. In such a way, these artificial SDF comments will be skipped by the parser. Figure 4.2 shows an example of such a transformation on a short portion of a COBOL program. Note that this pre-processing is reversible, in the sense that the resulting SDF comments can be easily replaced with COBOL comments during a post-processing phase.

Furthermore, in our project, we also had to deal with programs written in mixed-case letters. The COBOL IBM-VSII SDF grammar is written in upper-case. The `sglr` parser is case-sensitive, unlike the major part of the COBOL compilers. So we had to transform the programs from mixed-case keywords to upper-case keywords. Of course, we kept the comments and the strings in lower-case. Unlike the comments pre-processor, this pre-processor is not reversible. Note that the two pre-processor described above consist of two very simple Perl scripts.

Another typical pre-processing task is the *reformulation* [vSV97]. It consists of eliminating syntactic variations, without changing the semantics of the code. An example is to transform `GREATER OR EQUAL` to `>=`. There is a myriad of such syntactic freedoms in COBOL. By dealing them with a preprocessor, we can reduce the size of the COBOL SDF definition. In our project, we used the reformulation for other reasons. For instance, COBOL allows the programmer to open or close several files with a single statement. For instance we could find in a COBOL program the following open statement:

```
OPEN INPUT ORDERS STOCK
     I-O CUSTOMER
```

Remember that we will have to transform only the `OPEN` statements opening the files that have been migrated. So, using an ASF+SDF pre-processing phase, we *unfold* the `OPEN` and `CLOSE` statements. In this way the open statement above becomes three separate open statements, as follows:

```
[TagId]
 Left-hand side = * START NEW FRAGMENT
                    Right-hand side
                  * END NEW FRAGMENT

 when C1 , Cn , . . . , Cn
```

Figure 4.3: *Inserted comments for pretty-printing*

```
OPEN INPUT ORDERS
OPEN INPUT STOCK
OPEN I-O CUSTOMER
```

This pre-processing task is not reversible.

4.4 Pretty-printing

Once the COBOL program has been transformed, it must be printed correctly, especially because the COBOL layout is semantically relevant. The most important COBOL pretty-printing requirement is that the resulting program must be compilable. It means that each portion of the code has to be printed in an allowed *columns area*. For instance, statements must be written in columns 12-72 and each paragraph has to start with a paragraph-name beginning in column 8. The comments must start with an asterisk in column 7, etc. Furthermore, the transformed program must be recognizable by its authors. In order to achieve this goal, we must guarantee that the unchanged code is still printed (i.e, indented) in the same way as initially. This is the reason why we used the ASF+SDF interpreter instead of the faster ASF+SDF compiler. The interpreter, unlike the compiler, "*keeps the layout of sub-terms that are not rewritten*" [vV00]. So, the interpreter guarantees to keep the layout of the portions of the COBOL program that are not transformed. In our project, all the *non*-DMS statements are printed in the transformed program exactly in the same way as in the initial program. Moreover, the layout occurring in the right-hand side of a successfully equation is just left in the *normal* form, i.e., in the rewritten code. It means that we can control the layout of the rewritten pieces of code, except for the first line of the right-hand side that will be printed according to the layout of the initial fragment.

Our *local* pretty-printing approach[2] focusses on the rewritten fragments of the program. This approach consists of inserting special comments in the right-hand side of our equations in order to identify the rewritten fragments in the output program. We add a comment "`START NEW FRAGMENT`" at the beginning of the right-hand side and we add a comment "`END NEW FRAGMENT`" at the end of the right-hand side. The Figure 4.3 shows the general structure of our equations. The first asterisk in the rewritten sub-term is printed exactly in the same column as the initial sub-term. Using a Perl script, we identify the new fragments, that we indent based on the first asterisk. In Figure 4.4 and Figure 4.5 we show an example of the effect of our local pretty-printer. In Figure 4.4, we can see that the first line of the right-hand side (RHS) of the equation `* START NEW FRAGMENT` starts exactly in the same column as the rewritten `READ` statement (line 5 of Figure 4.4). The others lines (lines 6-22) are written by keeping the layout

[2]Many thanks to Steven Klusener and Niels Veerman for their support in this purpose.

occurring in the RHS, but they are not indented correctly. By applying our Perl script we obtain the code shown in Figure 4.5.

4.5 General Transformation Approach

Here we will present our general transformation approach and summarize the different steps of our COBOL source code transformations. This approach was used for both P1 and P2 implementations, discussed in Chapter 5 and Chapter 6 respectively. It could be used for other COBOL transformations as well.

Transformation process In Figure 4.6, we propose a general methodology for COBOL source code transformations. The transformation process is applied to a COBOL program, that we assume to be correct (`input.cob`). The result of the transformation is the COBOL program `output.cob`. `output.cob` has the same program logic as `input.cob`, but accesses the new SQL database instead of the migrated COBOL files.

The process starts with a pre-processing phase, discussed in Section 4.3. In our case, a Perl script is used to replace the COBOL comments with SDF comments. Note that the reformulation of the closing/opening of multiple files is integrated into the `Rewriting` phase.

The pre-processing phase is followed by a parsing phase. The parsing is based on the COBOL IBM-VSII SDF grammar, discussed in Section 4.1. The ASF+SDF Meta-Environment, presented in Chapter 3, allows to generate a parse table from this syntax definition. Based on the parse table, the `sglr` parser produces the syntax tree of the input program.

During the rewriting phase, the ASF conditional equations we defined are applied to the input program. In particular, the function `asfe` transforms the syntax tree of the input program into the rewritten syntax tree of the output program. Then, the latter is converted to a textual form.

The post-processing phase consists of converting the remaining SDF comments into COBOL comments in the transformed program.

Last, the pretty-printing phase, discussed in Section 4.4, prints the resulting program correctly. In particular, the output program `output.cob` can be compiled immediately.

```
1            IF REF-DET-STK(IND-DET) = 0
2                       MOVE 0 TO END-DETAIL
3                     ELSE
4                       MOVE REF-DET-STK(IND-DET) TO STK-CODE
5                       * START NEW FRAGMENT
6* READ STOCK
7        MOVE "read" TO SQL-ACTION
8        MOVE "KEY IS STK-CODE" TO SQL-OPTION
9        CALL "WR-STK"
10         USING SQL-ACTION
11               STK
12               SQL-OPTION
13               WR-STATUS
14* INVALID KEY
15       IF WR-STATUS-INVALID-KEY
16         THEN
17           DISPLAY "ERROR : UNKOWN PRODUCT"
18* NOT INVALID KEY
19         ELSE
20           DISPLAY STK-NAME "  " ORD-QTY(IND-DET)
21       END-IF
22* END NEW FRAGMENT
23                      SET IND-DET UP BY 1
```

Figure 4.4: *Before pretty-printing*

```
1            IF REF-DET-STK(IND-DET) = 0
2                       MOVE 0 TO END-DETAIL
3                     ELSE
4                       MOVE REF-DET-STK(IND-DET) TO STK-CODE
5      * READ STOCK
6                       MOVE "read" TO SQL-ACTION
7                       MOVE "KEY IS STK-CODE" TO SQL-OPTION
8                       CALL "WR-STK"
9                         USING SQL-ACTION
10                              STK
11                              SQL-OPTION
12                              WR-STATUS
13     * INVALID KEY
14                      IF WR-STATUS-INVALID-KEY
15                        THEN
16                          DISPLAY "ERROR : UNKOWN PRODUCT"
17     * NOT INVALID KEY
18                        ELSE
19                          DISPLAY STK-NAME "  " ORD-QTY(IND-DET)
20                      END-IF
21                      SET IND-DET UP BY 1
```

Figure 4.5: *After pretty-printing*

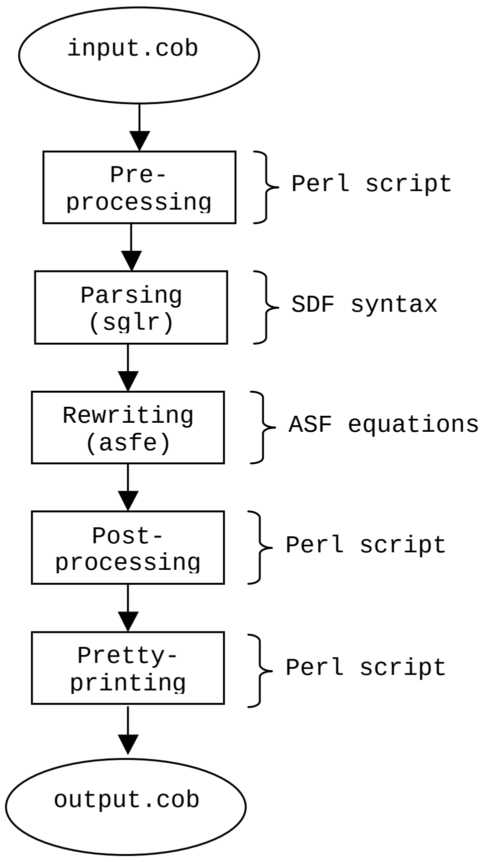


Figure 4.6: *COBOL Transformation process*

Chapter 5

Wrapper-based Implementation (P1)

In this Chapter, we present our ASF+SDF specification of the P1 COBOL program conversion strategy, following a D2 database conversion strategy.

We assume that for each COBOL file that have been migrated, an inverse wrapper has been generated. Each wrapper allows the legacy programs to access a migrated COBOL file in the new SQL database, by providing them a *legacy* interface.

As already mentioned, the P1 conversion consists of replacing the DMS COBOL statements accessing the migrated files, with wrapper invocations. The naming convention we used is the following: the name of a wrapper has the form "`WR-`*record-type*", where *record-type* is the name of the record type of the migrated file.

5.1 Useful structures and functions

In our P1 implementation, we frequently used some ASF+SDF structures and functions. This is the reason why we will start this chapter by describing them.

Table

The generic structure `Table[[Key,Value]]` is a kind of hash table, provided in the ASF+SDF library. This built-in provides the usual hash table operators, including:

- `store(Table[[Key,Value]],`*key*`,`*value*`)`: stores a relation $<$ *key*,*value* $>$ in the table
- `lookup(Table[[Key,Value]],`*key*`)`: returns the *value* that corresponds to the given *key*
- `element(Table[[Key,Value]],`*key*`)`: returns *true* if the table contains a relation with the key value *key*, else it returns *false*
- `[]`: represents the empty table

In our project, we have to keep track of relations between file names and record type names, as well as relations between file names and primary key names. These COBOL names have got the same low-level lexical sort: `Lex-cobword`. Thus, our specification uses the `Table[[Key,Value]]` container, where both `Key` and `Value` are `Lex-cobword`.

Mylist

The structure `Mylist`, that we implemented, is a list of `Lex-cobword`. Such a list has the following form:

`mylist(`$cobword_1$`,`$cobword_2$`,...,`$cobword_n$`)`

The sort `Mylist` provides the usual functions of a list, including:

- `add(`*mylist*`,`*cobword*`)`: add the element *cobword* to the list
- `elemt(`*mylist*`,`*cobword*`)`: returns *true* if *cobword* is in the list, else it returns *false*

add-comment

The function `add-comment` transforms a sequence of characters into a COBOL comment. For instance "`add-comment("hello world")`" will be rewritten as the comment "`*hello world`". Here is the signature of this function:

```
    add-comment(Nonnumeric-dq) -> ExplicitComment
```

open-and-close-unfolder

As explained in Section 4.3, we need to perform a *reformulation* pre-processing phase, to *unfold* the `OPEN` and `CLOSE` statements. We defined a transformer traversal function, called `open-and-close-unfolder`. Here is the SDF declaration of this function:

```
    open-and-close-unfolder(Cobol-source-program)
        -> Cobol-source-program {traversal(trafo)}
```

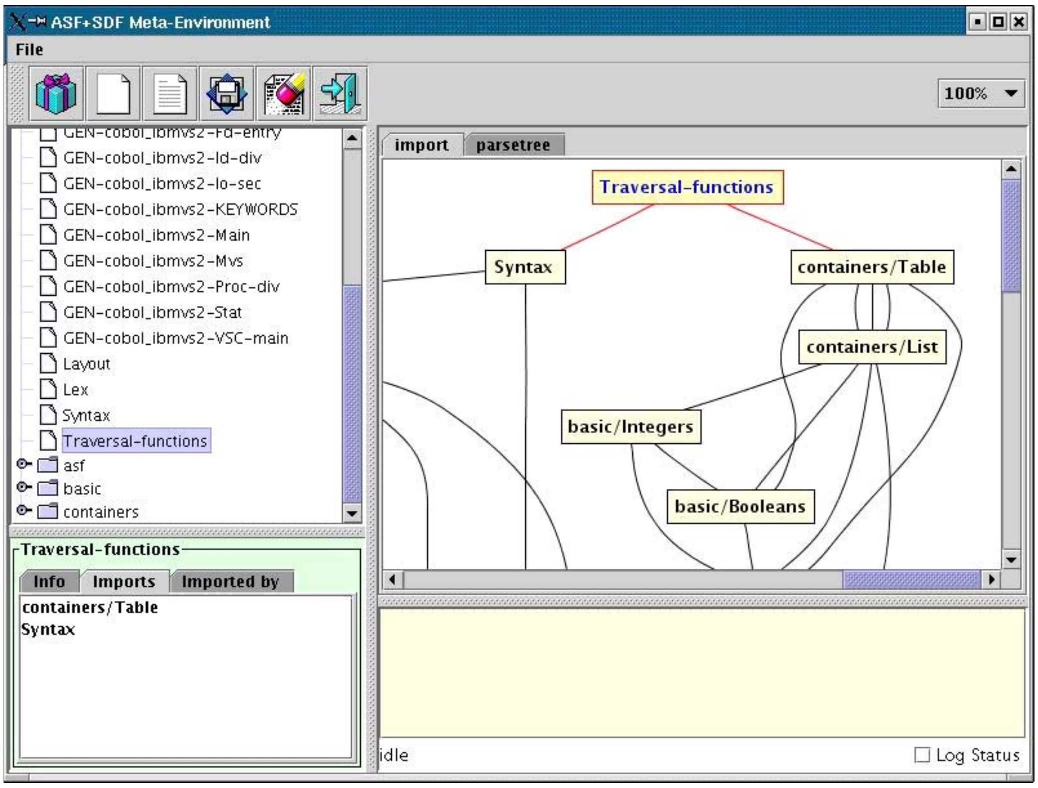
Here is one of its equations, unfolding the `CLOSE` statement:

```
[close-unfolder]
    open-and-close-unfolder(statement*1
                            CLOSE file-name1 file-name+1
                            statement*2)
        = statement*3
          CLOSE file-name1
          open-and-close-unfolder(CLOSE file-name+1)
          statement*4
    when statement*3 = open-and-close-unfolder(statement*1),
         statement*4 = open-and-close-unfolder(statement*2)
```

5.2 Overview

Our P1 ASF+SDF specification consists of a module, called *Traversal-functions*, that imports:

- the IBM-VSII COBOL SDF syntax definition
- the module `Table[[Lex-cobword,Lex-cobword]]`



containers/Table
Syntax

Figure 5.1: *Module imports graph*

Figure 5.1 shows the module imports graph, as it can be seen in the user interface of the ASF+SDF Meta-Environment. The module *Traversal-functions* defines the P1 conversion, consisting of several traversal functions. First, we wrote accumulator functions, allowing us to return some results during the traversal of the COBOL program. Thanks to these accumulators, we are able to retrieve the information we need from the program, in order to transform the latter. The P1 conversion itself is performed by transformer functions, that use the results of the accumulator functions.

5.2.1 Accumulators

For the P1 conversion of a COBOL program, we particularly need to know:

1. the record type name of each migrated file defined in the program
2. the primary key name of each migrated file defined in the program
3. the record declarations of the migrated files defined in the program

So we defined the three corresponding accumulators as follows:

record-name-table

```
record-name-table(Cobol-source-program,
                  Table[[Lex-cobword,Lex-cobword]],
                  Mylist)
   -> Table[[Lex-cobword,Lex-cobword]] {traversal(accu)}
```

record-key-table

```
record-key-table(Cobol-source-program,
                 Table[[Lex-cobword,Lex-cobword]],
                 Mylist)
   -> Table[[Lex-cobword,Lex-cobword]] {traversal(accu)}
```

datalist

```
datalist(Cobol-source-program,
         Data-description-entry*,
         Mylist)
   -> Data-description-entry* {traversal(accu)}
```

Remember that the first argument of each accumulator is the traversed tree, i.e., a COBOL program syntax tree. The second argument is the resulting accumulation value, e.g., a hash table of sort `Table[[Lex-cobword,Lex-cobword]]`. The third argument, of sort `Mylist`, is the list of the COBOL files that have been migrated into SQL tables.

5.2.2 Transformers

Once these three results have been built, we can use them as additional parameters of our transformers. Allowing us to pass additional parameters is indeed one of the greatest merits of the traversal functions. Let us recall what the P1 conversion does change in a COBOL program:

1. removing the *SELECT* clause of the migrated files from the `INPUT-OUTPUT` section in the `ENVIRONMENT` division

2. moving the record declarations of the migrated files from the FILE section to the `WORKING-STORAGE` section in the `DATA` division

3. replacing each access statement with a wrapper invocation in the `PROCEDURE` division

We can consider that there are two separate transformations to perform. Firstly, we have to reorganize the `ENVIRONMENT` and `DATA` divisions. Secondly, we have to rewrite the access statements accessing the migrated files in the `PROCEDURE` division. So we defined two transformers, namely `file-to-working` and `access-to-call`.

file-to-working

```
file-to-working(Cobol-source-program,
                Data-description-entry*,
                Mylist)
   -> Cobol-source-program {traversal(trafo)}
```

The second argument of `file-to-working` is the sequence of record declarations constructed by the accumulator `datalist`. The third argument is the list of the COBOL files that have been migrated into SQL tables.

The resulting program of

`file-to-working`($program_1, record\text{-}declarations, files\text{-}list$)

is the same program as $program_1$, where:

1. the `SELECT` clauses of the migrated files have been removed from the `INPUT-OUTPUT` section of the `ENVIRONMENT` division

2. the record declarations of the migrated files have been moved from the `FILE` section to the `WORKING-STORAGE` section

The second argument *record-declarations* is the sequence of record declarations constructed by the accumulator `datalist`.

access-to-call

```
    access-to-call(Cobol-source-program,
                   Table[[Lex-cobword,Lex-cobword]],
                   Table[[Lex-cobword,Lex-cobword]])
       -> Cobol-source-program {traversal(trafo)}
```

The second and third arguments of the function `access-to-call` are respectively the record-name table and the record-key table of the program to transform.

The resulting program of

`access-to-call`$(program_1, rn\text{-}table, rk\text{-}table)$

is the same program as $program_1$, where all the DMS statements accessing a migrated file have been replaced with wrapper invocations. The second argument *rn-table* is the table constructed by the accumulator `record-name-table`. The third argument *rk-table* is the table constructed by the accumulator `record-key-table`.

5.3 Selected Equations

In this section, we will explain some of our equations. Note that some of them were simplified in order to be more readable. We will start with the top-level P1 transformation equation, before discussing in details the traversal functions it contains.

5.3.1 P1 transformation

The global p1 transformation consists of applying our two transformers on the COBOL program. This is exactly what the function `finaltransfo` does:

```
[finaltrans]
 finaltransfo(program1, file-list1) = program4

 when
   datadec1* = datalist(program1,
                           01 WR-STATUS PIC 9(3).
                             88 WR-STATUS-NO-ERR VALUE 0.
                             88 WR-STATUS-INVALID-KEY VALUE 1.
                             88 WR-STATUS-AT-END VALUE 100.

                           01 SQL-ACTION PIC X(100).

                           01 SQL-OPTION PIC X(100).,

                           file-list1),

   rn-table = record-name-table(program1,[],file-list1),
   rk-table = record-key-table(program1,[],file-list1),
   program2 = file-to-working(program1,datadec1*,file-list1),
   program3 = open-and-close-unfolder(program2),
   program4 = access-to-call(program3,rn-table,rk-table)
```

In the equation above, we produce the transformers' arguments, by applying the accumulators. We build the record-name table (`rn-table`), the record-key table (`rk-table`) and the sequence of the record declarations (`datadec1*`) of the migrated files (listed in `file-list1`).

We apply our transformers functions successively on the initial program (`program1`). We first apply the `open-and-close-unfolder` function. Then, we reorganize the `ENVIRONMENT` and the `DATA` divisions, by applying `file-to-working`. Last, we replace the DMS statements (accessing the migrated files) with wrapper invocations, by using `access-to-call`.

Note that we can give some additional data declarations, as the initial value of the accumulating argument of the `datalist` function. In this way, these declarations will be added, with the record declarations, in the `WORKING-STORAGE` section, by the function `file-to-working`. The variable `WR-STATUS` will be used as a wrapper invocation argument. It will contain the state of the called wrapper. For example, it will indicate when the program tries to read the "file" with an invalid access key. `SQL-ACTION` and `SQL-OPTION` will be two other arguments of the wrapper invocations. They will contain the access type to perform (read, write,..) and the option of this access, respectively.

The resulting COBOL program (`program4`) does not contain the migrated files declarations anymore, and invokes wrappers instead of accessing these files.

5.3.2 Accumulators

record-name-table

```
[record-name-table]
  record-name-table(FD file-name1.
                   01 record-name1.
                      datadec1* , table1, file-list1)
    = store(table1,file-name1,record-name1)

   when elemt(file-list1, file-name1) = true
```

In this equation, we store in the accumulating argument (`table1`) the correspondence between a file and its record type name. We can retrieve this information, by analyzing a file description. The description of each file starts with the name of the file (`file-name1`), followed by some optional clauses, and declares the record type of the file (`record-name1`) followed by all its sub-level fields description (`data-dec1*`).

We have to store the relation between the file (`file-name1`) and its record type (`record-name1`), in the resulting table. Note that we store the relation in the table, only if `file-name1` has been migrated, i.e., when `elemt(file-list1,file-name1)` is true.

Example Let us consider a program containing the `FILE` section shown in Figure 5.2. If all the files have been migrated, the resulting `record-name-table` is:

```
[<CUSTOMER,CUS>,<ORDERS,ORD>,<STOCK,STK>]
```

```
FILE SECTION.

  FD CUSTOMER.
   01 CUS.
     02 CUS-CODE PIC X(12).
     02 CUS-DESCR PIC X(110).
     02 CUS-HIST PIC X(1000).

  FD ORDERS.
   01 ORD.
     02 ORD-CODE PIC 9(10).
     02 ORD-DATE PIC X(8).
     02 ORD-CUSTOMER PIC X(12).
     02 ORD-DETAIL PIC X(200).

  FD STOCK.
   01 STK.
     02 STK-CODE PIC 9(5).
     02 STK-NAME PIC X(100).
     02 STK-LEVEL PIC 9(5).
```

Figure 5.2: *Example of a* `FILE` *section*

Left-hand side of the equation	*File declaration*
`FD file-name1.`	`FD ORDERS.`
`01 record-name1`	`01 ORD.`

`datadec1*`	`02 ORD-CODE PIC 9(10).`
	`02 ORD-DATE PIC X(8).`
	`02 ORD-CUSTOMER PIC X(12).`
	`02 ORD-DETAIL PIC X(200).`

Figure 5.3: *Matching of the* `record-name-table` *equation*

The equation `[record-name-table]` has been applied three times during the traversal of the program. The three `FD` paragraphs of Figure 5.2 do match with the left-hand side of that equation. For instance, the Figure 5.3 shows the matching between the `FD` paragraph of the file `ORDERS` and the `FD` paragraph of the left-hand side of the equation `[record-name-table]`. The variables of the left-hand side can be replaced by terms in such a way that the result is precisely the parsed FD paragraph.

record-key-table

```
[record-key-table]
   record-key-table(SELECT file-name assign-clause
                       fce-phrase*1
                       RECORD key? is? record-key
                       fce-phrase*2 .   , table1, file-list1)
   = store(table1,file-name,record-key)

   when elemt(file-list1, file-name) = true
```

By parsing a `SELECT` clause, we can figure out what is the primary key of a file. For a sequential or an indexed file, the `SELECT` clause contains a `RECORD KEY` phrase, indicating

```
FILE-CONTROL.

   SELECT CUSTOMER ASSIGN TO "c:\CUSTOMER.DAT"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS CUS-CODE.

   SELECT ORDERS ASSIGN TO "c:\ORDERS.DAT"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS ORD-CODE
       ALTERNATE RECORD KEY IS ORD-CUSTOMER
         WITH DUPLICATES

   SELECT STOCK ASSIGN TO "c:\STOCK.DAT"
       ORGANIZATION IS INDEXED
       ACCESS MODE IS DYNAMIC
       RECORD KEY IS STK-CODE.
```

Figure 5.4: *Example of a* `FILE-CONTROL` *section*

Left-hand side of the equation	*File declaration*
`SELECT file-name`	`SELECT ORDERS`
`assign-clause`	`ASSIGN TO "c:\ORDERS.DAT"`

`fce-phrase*1`	`ORGANIZATION IS INDEXED`
	`ACCESS MODE IS DYNAMIC`

`RECORD key? is? record-key`	`RECORD KEY IS ORD-CODE`

`fce-phrase*2`	`ALTERNATE RECORD KEY IS ORD-CUSTOMER`
	`WITH DUPLICATES`

Figure 5.5: *Matching of the* `record-key-table` *equation*

the name of the record field used as primary key. We simply store in the accumulating table the relation between the file (`file-name`) and its primary key (`record-key`). For a relative file, the `RELATIVE KEY` phrase indicates the data item used as primary key. We wrote a second similar equation of the `record-key-table` accumulator, using the `RELATIVE KEY` phrase instead of the `RECORD KEY` phrase.

Example Let us consider a program containing the `FILE-CONTROL` section shown in Figure 5.4 in its `ENVIRONMENT` division: If all the files have been migrated, the resulting `record-key-table` is:

```
[<CUSTOMER,CUS-CODE>,<ORDERS,ORD-CODE>,<STOCK,STK-CODE>]
```

The equation `[record-key-table]` has been applied three times during the traversal of the program. The three `SELECT` clauses above do match with the left-hand side of that equation. For instance, the Figure 5.5 shows the matching between the `SELECT` clause of the file `ORDERS` and the `SELECT` clause of the left-hand side of the equation `[record-key-table]`.

Left-hand side of the equation	*File declaration*
`FD file-name1.`	`FD ORDERS.`

`datadec1+`	`01 ORD.`
	` 02 ORD-CODE PIC 9(10).`
	` 02 ORD-DATE PIC X(8).`
	` 02 ORD-CUSTOMER PIC X(12).`
	` 02 ORD-DETAIL PIC X(200).`

Figure 5.6: *Matching of the* `datalist` *equation*

datalist

```
[datalist]
   datalist(FD file-name1.
              datadec1+, datadec2*, file-list1)
   =  datadec1+
      datadec2*
   when elemt(file-list1, file-name1) = true
```

Each entry of the `FILE` section, contains the declaration of the record type of the file (`datadec1+`). If the file has been migrated, we add its record data declaration to the accumulating argument. Thus, the returning value of the function is the sequence of record declarations.

Example Let us consider a program containing the `FILE` section shown in Figure 5.2, that describes three files. If these three files have been migrated, the equation `[datalist]` is applied three times. The resulting value consists of the three corresponding record declarations. The Figure 5.6 shows the matching between the `FD` paragraph of the file `ORDERS` and the `FD` paragraph of the left-hand side of the equation `[datalist]`.

5.3.3 Transformers

file-to-working The function `file-to-working` reorganizes the `ENVIRONMENT` and `DATA` divisions.

First, it removes the `SELECT` clause of each migrated file from the `INPUT-OUTPUT` section of the `ENVIRONMENT` division (Figure 5.7). Second, it removes the `FD` paragraph of each migrated file from the `FILE` section of the `DATA` division (Figure 5.8). Last, it declares the record descriptions of the migrated files, and the other additional variables, in the `WORKING-STORAGE` section (Figure 5.9).

access-to-call Since it would be impossible to include here all the equations of `access-to-call`, we have selected only one of them. The latter is shown in Figure 5.10. In order to make it more readable, we did not separate each character in the constructor functions, as we normally should.

```
[file-to-working-environment]
    file-to-working(file-control-entry*1
                    SELECT file-name1 assign-clause
                    fce-phrase*1.
                    file-control-entry*2 ,datadec1*,file-list1)
    = file-to-working(file-control-entry*1,datadec1*,file-list1)
      file-to-working(file-control-entry*2,datadec1*,file-list1)

    when elemt(file-list1, file-name1) = true
```

Figure 5.7: *Equation 1 of* `file-to-working`

```
[file-to-working-file-section]
    file-to-working(file-section-entry*1
                    FD file-name1.
                       datadec1+
                    file-section-entry*2,datadec1*,file-list1)
     = file-to-working(file-section-entry*1,datadec1*,file-list1)
       file-to-working(file-section-entry*2,datadec1*,file-list1)

    when elemt(file-list1, file-name1) = true
```

Figure 5.8: *Equation 2 of* `file-to-working`

```
[file-to-working-working-storage]
    file-to-working(WORKING-STORAGE SECTION.
                  datadec2*,datadec1*)
     = WORKING-STORAGE SECTION.
         datadec1*
         datadec2*
```

Figure 5.9: *Equation 3 of* `file-to-working`

```
[read-invalid-key]
    access-to-call(statement*1
                   READ file-name1
                    INVALID key?
                       statement-list1
                    NOT INVALID key?
                       statement-list2
                   END-READ
                   statement*2, rn-table, rk-table)
    =    statement*3
      add-comment(access-statement)
         MOVE "read" TO SQL-ACTION
         MOVE nonnumeric2 TO SQL-OPTION
         CALL nonnumeric1 USING
           BY CONTENT SQL-ACTION
           BY REFERENCE record-name1
           BY CONTENT SQL-OPTION
           BY REFERENCE WR-STATUS
      add-comment("INVALID KEY")
         IF WR-STATUS-INVALID-KEY
           THEN
               statement-list1
      add-comment("NOT INVALID KEY")
           ELSE
               statement-list2
         END-IF
         statement*4

 when
    file-name1 = lex-cobword(char+1)
    record-name1 = lookup (rn-table,file-name1),
    record-name1 = lex-cobword(char+2),
    nonnumeric1 = nonnumeric-dq("WR- char+2"),
    record-key1 = lookup(rk-table,file-name1),
    record-key1 = lex-cobword(char+3),
    nonnumeric2 = nonnumeric-dq("KEY IS char+3"),
    access-statement = nonnumeric-dq("READ char+1 KEY IS char+3"),
    statement*3 = access-to-call(statement*1, rn-table, rk-table),
    statement*4 = access-to-call(statement*2, rn-table, rk-table)
```

Figure 5.10: *One equation of* access-to-call

This equation replaces a random `READ` statement, located in a sequence of statements, with a `CALL` to a wrapper. Remember that the arguments of such an invocation are respectively:

- the sql action to perform (`SQL-ACTION`)
- the name of the record type of the file (`record-name1`)
- the option of the access (`SQL-OPTION`)
- the wrapper status (`WR-STATUS`)

In the context of a random `READ`, the sql action is "read", so we write the statement: `MOVE "read" TO SQL-ACTION`. The sql option has the form: `"KEY IS` *primary-key*`"`. Indeed, this random `READ` does not specify an access key. Thus, the read has to be performed using the primary key of the file. Thanks to the `rk-table`, we are able to lookup the name of the primary key of the file (`record-key1`). In the same way, we can retrieve the name of the record type of the file (`record-name1`).

Note that we use lexical constructor functions to get access to the characters of several tokens and to construct other ones. We apply `access-to-call` to the sequences of statements that precede and follow the random `READ`. These sequences are represented by the variable `statement*1` and `statement*2`, respectively.

The function `add-comment` allows us to add a comment in the source code, by giving its content as an argument. In this way, we can keep the trace of the rewritten statements, by rewriting them as COBOL comments.

Example Let `CUSTOMER` be a file. Its record type is `CUS` and its primary record key is `CUS-CODE`. The file `CUSTOMER` has been migrated through a D2 database conversion phase. A wrapper has been written (`WR-CUS`) to provide a legacy interface for accessing the migrated file `CUSTOMER`. Figure 5.11 shows the P1 transformation of a random `READ` of the file `CUSTOMER`, using the `access-to-call` function.

```
READ CUSTOMER
   INVALID KEY
      DISPLAY "ERROR"
      MOVE 1 TO END-FILE
   NOT INVALID KEY
      PERFORM NEW-ORDER
END-READ
```

a) before transformation

```
*READ CUSTOMER
  MOVE "read" TO SQL-ACTION
  MOVE "KEY IS CUS-CODE" TO SQL-OPTION
  CALL WR-CUS USING
    BY CONTENT SQL-ACTION
    BY REFERENCE CUS
    BY CONTENT SQL-OPTION
    BY REFERENCE WR-STATUS
*INVALID KEY
  IF WR-STATUS-INVALID-KEY
    THEN
        DISPLAY "ERROR"
        MOVE 1 TO END-FILE
*NOT INVALID KEY
    ELSE
        PERFORM NEW-ORDER
  END-IF
```

b) after transformation

Figure 5.11: *Effect of* `access-to-call`

Chapter 6

Statement Rewriting Implementation (P2)

In this Chapter, we present our ASF+SDF specification of the P2 COBOL program conversion strategy, following a D1 database conversion strategy, discussed in Section 6.1. Remember that in this work the P2 COBOL conversion consists of replacing the DMS COBOL statements accessing the migrated files, with the corresponding SQL queries.

As seen before, we decided to generate the SQL code translating the COBOL DMS statements in a separate section of the program. The so-called `P2-STRATEGY` section contains several P2 paragraphs accessing the new SQL database. The generation of those paragraphs is discussed in details in Section 6.3.

6.1 D1 conversion strategy

The D1 database conversion strategy does not change the structure of the database. We previously said that the D1 conversion strategy translates each COBOL record-type into a SQL table and each top-level field into a column of this table. Actually, this is only the D1 *default* rule. Let consider the file `STUDENT` declare in Figure 6.1. In this case, the D1 *default* rule should not be applied. If we only translate the three top-level fields into three SQL columns, we would not be able to define `LAST-NAME` as a SQL

```
SELECT STUDENT                        FD STUDENT.
  ASSIGN TO "c:\student"              01 STUD.
  ORGANIZATION IS INDEXED               02 NAME.
  ACCESS MODE IS DYNAMIC                  03 FIRST-NAME PIC X(12).
  RECORD KEY IS NAME                      03 LAST-NAME PIC X(16).
  ALTERNATE RECORD KEY IS LAST-NAME     02 ADDRESS.
    WITH DUPLICATES                       03 STREET PIC X(20).
                                          03 NUMBER PIC X(5).
                                          03 CITY PIC X(10).
                                        02 SCHOOL PIC X(20).
```

Figure 6.1: *Definition of the file* `STUDENT`

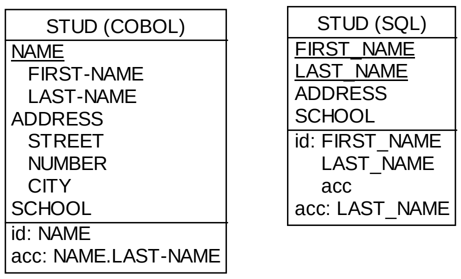


Figure 6.2: *Example of a D1 conversion*

access key.

Thus, when a top-level field contains at least one sub-level field defined as an access key, it is *disaggregated* before being translated into SQL. The requirement is that each COBOL access key must become at least one SQL column in order to be declared as a SQL access key. The record description above would be translated into a table of four columns, as shown in Figure 6.2. As we can see, the top-level field `NAME` has been disaggregated. Its two sub-level fields are translated into two SQL columns during the D1 conversion. In this way, we can declare both the record key and the alternate key. The first one consists of two columns.

6.2 Additional variables

In the following section, "*P2 generated paragraphs*", we will use several additional variables in order to translate the COBOL DMS statements into the corresponding SQL statements. As we saw in the previous chapter, we can easily add new variables declarations in the `WORKING-STORAGE` section[1]. In this section, we will describe the additional variables we need in our P2 implementation.

SQLCODE The `SQLCODE` is used as a status returning value of a SQL operation. For instance, when its value is zero, it means that the operation was performed successfully. The variable `SQLCODE` is one of the fields of the `SQLCA` structure. This structure can be included in the `WORKING-STORAGE` section as follows:

```
EXEC SQL INCLUDE SQLCA END-EXEC
```

P2-COUNTER The variable `P2-COUNTER` is a simple counter, used as the resulting value of a SQL `SELECT COUNT(*)` clause. For instance, by testing its value, we are able to know the number of rows from a table having a given key value. If this number is zero, there is an `INVALID KEY` exception.

[1]by using the initial value of the accumulator `datalist` and the transformer `file-to-working`

P2-STATUS Remember that we used, in the previous chapter, the variable `WR-STATUS` to keep the status of the called wrappers. In the case of P2, the transformed program directly accesses the SQL database. The variable `P2-STATUS` will indicate if the accesses are performed successfully. By using `P2-COUNTER` and `SQL-CODE`, we can detect the `INVALID KEY` and the `AT END` exceptions, and simulate them by modifying the value of `P2-STATUS`. In such a way, the `INVALID KEY` and the `AT END` phrases can be replaced with a test of the `P2-STATUS` value. Here is the declaration of this variable:

```
01 P2-STATUS PIC S9(9).
  88 P2-STATUS-NO-ERR VALUE 0.
  88 P2-STATUS-INVALID-KEY VALUE 1.
  88 P2-STATUS-AT-END VALUE 100.
  88 P2-STATUS-SQL-ERROR VALUE 99.
```

Last cursor variables In order to simulate the `START/READ NEXT` sequence with a cursor based loop, we need to store the current open cursor on each table of the new database. So, for each table we declare one variable having the form "`LAST-CURSOR-`*table*".

6.3 P2 generated paragraphs

In this section, we will define as precisely as possible what the P2 paragraphs contain, i.e., how we can translate COBOL DMS statements into SQL DMS statements.

Our starting point is the D1 conversion of a COBOL file. The D1 conversion does not change the data structure, but just translates it in the new DML. Thus, there is a one-to-one mapping between the legacy COBOL files and the new SQL tables. This mapping can be defined as follows:

* Let $file$ be a COBOL file

* Let $record$ be its record type

* Let *table* be the D1 SQL translation of $record$

* *table* contains n columns, called $c_1, c_2, ..c_n$, that translates the fields $f_1, f_2, ..f_n$ of $record$. When $file$ is a relative file, one column c_i translates the `RELATIVE KEY` as well.

* $file$ has one primary key, say *prim-key*.

* $file$ has m `ALTERNATE RECORD KEY`s, called $ak_1, ak_2, ..ak_m$

* Each access key (*prim-key* or ak_i) has been translated into one or many columns of *table*.

6.3.1 Cursor declarations paragraph

As previously mentioned, we will use SQL cursors. All the SQL cursors declarations are generated in a COBOL paragraph in the `P2-STRATEGY` section. For each access key (*prim-key* or ak_i), we declare three SQL cursors: the "*order by*" cursor, the "*greater than*" cursor and the "*not less*" cursor.

Order by cursor

The order by cursor has the following general form:

```
EXEC SQL
  DECLARE cursor-name CURSOR FOR
    SELECT $c_1$,$c_2$,...,$c_n$
    FROM table
    ORDER BY $ck_1$,$ck_2$,...,$ck_p$
END-EXEC
```

where ck_1,ck_2,...,ck_p are the SQL columns the access key has been translated into[2]

Example Here is the "*order by*" cursor declaration for the access key `NAME` described in Figure 6.1. In this case, $p = 2$:

```
EXEC SQL
  DECLARE ORDER_NAME CURSOR FOR
    SELECT FIRST_NAME,LAST_NAME,ADDRESS,SCHOOL
    FROM STUD
    ORDER BY FIRST_NAME,LAST_NAME
END-EXEC
```

greater than cursor

The greater than cursor has the following general form:

```
EXEC SQL
  DECLARE cursor-name" CURSOR FOR
    SELECT $c_1$,$c_2$,...,$c_n$
    FROM table
    WHERE  ($ck_1$ > $fk_1$) OR
           (($ck_1$ = $fk_1$) AND ($ck_2$ > $fk_2$)) OR
              ...
           (($ck_1$ = $fk_1$) ... AND ($ck_{p-1}$ = $fk_{p-1}$) AND ($ck_p$ > $fk_p$))
    ORDER BY $ck_1$,$ck_2$,...,$ck_p$
END-EXEC
```

where fk_1,fk_2,...,fk_p are the disaggregated fields of the access key, translated into ck_1,ck_2,...,ck_p

[2]Most of the time, $p = 1$

Example

```
EXEC SQL
  DECLARE GREATER_NAME CURSOR FOR
    SELECT FIRST_NAME,LAST_NAME,ADDRESS,SCHOOL
    FROM STUD
    WHERE (FIRST_NAME > :FIRST-NAME) OR
          ((FIRST_NAME = :FIRST-NAME) AND (LAST_NAME > :LAST-NAME))
    ORDER BY FIRST_NAME,LAST_NAME
END-EXEC
```

Remark Note that the names of the COBOL fields can be duplicated in several record type descriptions. For instance, the name of the sub-level field `FIRST-NAME` could be used in another record type such as `TEACHER`. This is the reason why we should use the complete "path-names" of the fields used in the cursor declarations. In the case of the example above, we should replace `":FIRST-NAME"` with `":STUD.NAME.FIRST-NAME"` and `":LAST-NAME"` with `":STUD.NAME.LAST-NAME"`. This is implemented correctly in our P2 implementation. But, we will not use the complete names in our examples, in order to make them more readable.

not less cursor

The *not less* cursor has the same form as the *greater than* cursor. But each ">" is replaced with ">=".

6.3.2 Last cursor closing paragraphs

We said previously that before a SQL cursor can be opened, the current open cursor used on the same table must be closed. So we generate, for each SQL table, a COBOL paragraph that tests which is the current cursor of the table, and closes the latter. As already mentioned, we declared several variables having the form `LAST-CURSOR-`*table*. A last cursor paragraph simply tests the value of the corresponding variable and closes the current cursor.

The generation of these paragraphs can be done correctly only if:

1. We define a naming convention for the cursors.
2. We respect this convention everywhere in the generated code.
3. When a cursor is open, its name is stored in the corresponding `LAST-CURSOR-`*table* variable.

The Figure 6.3 shows an example of such a paragraph, in the case of the SQL table `STUD` in Figure 6.2. We can see that we test the six possible values of the variable `LAST-CURSOR-STUD`, that are the result of combining two access keys with three key usages.

```
P2-CLOSE-LAST-CURSOR-STUD.
   IF (LAST-CURSOR-STUD = "ORDER_BY_NAME")
        EXEC SQL
           CLOSE ORDER_BY_NAME
        END-EXEC.
   IF (LAST-CURSOR-STUD = "GREATER_NAME")
        EXEC SQL
           CLOSE GREATER_NAME
        END-EXEC.
   IF (LAST-CURSOR-STUD = "NOT_LESS_NAME")
        EXEC SQL
           CLOSE NOT_LESS_NAME
        END-EXEC.
   IF (LAST-CURSOR-STUD="ORDER_BY_LAST_NAME")
        EXEC SQL
           CLOSE ORDER_BY_LAST_NAME
        END-EXEC.
   IF (LAST-CURSOR-STUD="GREATER_LAST_NAME")
        EXEC SQL
           CLOSE GREATER_LAST_NAME
        END-EXEC.
   IF (LAST-CURSOR-STUD="NOT_LESS_LAST_NAME")
        EXEC SQL
           CLOSE NOT_LESS_LAST_NAME
        END-EXEC.
```

Figure 6.3: *A closing last cursor P2 paragraph*

6.3.3 OPEN paragraphs

We saw that the result of the COBOL `OPEN` statement depends on the opening option. This option can be `INPUT`, `OUTPUT`, `I-O` or `EXTEND`. These distinct opening modes can be easily translated into SQL. Indeed, SQL allows the user to read, update and add records, i.e., rows.

There is a difference between the open option `OUTPUT` and the other ones. When a COBOL file is opened with the `OUTPUT` mode, it is created (if necessary) and positioned to its starting point for writing. It means that if the file already exists, it is overwritten. For the other open options, the file is either read or updated.

Another important aspect is the COBOL access mode of the migrated file. When the access mode is `SEQUENTIAL`, the records are read in the ascending order based on their primary key. When the access mode is `RANDOM` or `DYNAMIC`, COBOL assumes the primary key of the file is used if the programmer does not supply an access key.

Thus, in all the cases, the first `"READ` *file*`"` statement of the program execution reads the first record of the file having the minimal primary key value. This is the reason why a COBOL `OPEN` can be translated into a cursor opening. This cursor is the *order-by* cursor corresponding to the primary key of the table. When the open option is `OUTPUT`, we first delete the content of the table. Figure 6.4 shows the P2 open paragraph for the file `STUDENT` discussed above. The Figure 6.5 shows the same open paragraph, but with the `OUTPUT` open option.

```
P2-OPEN-STUDENT.
    PERFORM P2-CLOSE-LAST-CURSOR-STUD.
    MOVE "ORDER_BY_NAME" TO LAST-CURSOR-STUD.
    EXEC SQL
        OPEN ORDER_BY_NAME
    END-EXEC.
```

Figure 6.4: `OPEN` *paragraph for the file* `STUDENT`

```
P2-OPEN-OUTPUT-STUDENT.
    PERFORM P2-CLOSE-LAST-CURSOR-STUD.
    MOVE "ORDER_BY_NAME" TO LAST-CURSOR-STUD.
    EXEC SQL
        DELETE FROM STUD
    END-EXEC.
    EXEC SQL
        OPEN ORDER_BY_NAME
    END-EXEC.
```

Figure 6.5: `OPEN OUTPUT` *paragraph for the file* `STUDENT`

6.3.4 START paragraphs

Remember that the `START` statement positions an indexed or relative file to a specific record. This can be translated in SQL by a cursor opening on the corresponding table. There are three key usages that can be used with the start statement: *equal*, *greater than* and *not less*. Since the `START` can be executed with an `INVALID KEY`, we first have to test if there exists at least one row in the table for which the key value is *equal to*/*greater than*/*not less than* the current value of the supplied key. If there is no such row in the table, we have to simulate an `INVALID KEY` exception, by modifying the variable `P2-STATUS`. If there is at least one such row, we can open the corresponding cursor. The Figure 6.6 shows which kind of SQL cursor is opened in order to simulate each `START` key usage. The Figure 6.7 shows the `START GREATER` paragraph for the file `STUDENT` describe in Figure 6.1.

6.3.5 READ NEXT paragraphs

The sequential `READ` can be translated into a SQL `FETCH` of the current opened cursor. Using the corresponding `LAST-CURSOR` variable, we can figure out which is the name of the current opened cursor on the target table, and fetch that cursor. The `INTO` clause

key usage	**cursor**
equal	*not less*
greater than	*greater than*
not less	*not less*

Figure 6.6: `START` *key usages VS SQL cursors*

```
P2-START-STUDENT-GREATER-NAME.
    PERFORM CLOSE-LAST-CURSOR-STUD
    EXEC SQL
       SELECT COUNT(*)
       INTO :P2-COUNTER
       FROM STUD
       WHERE (FIRST_NAME > FIRST-NAME) OR
             ((FIRST_NAME = FIRST-NAME) AND
              (LAST_NAME > LAST-NAME))
    END-EXEC
    IF (SQLCODE NOT = 0) %% SQL error
      SET P2-STATUS-SQL-ERROR TO TRUE
    ELSE
      IF (P2-COUNTER = 0) %% invalid key !
         SET P2-STATUS-INVALID-KEY TO TRUE
      ELSE
         EXEC SQL
           OPEN GREATER_NAME
         END-EXEC
         MOVE "GREATER_NAME"
              TO LAST-CURSOR-STUD
         MOVE SQLCODE TO P2-STATUS
      END-IF
    END-IF.
```

Figure 6.7: `START` *paragraph for the file* `STUDENT`

of the `FETCH` query contains the fields that have been translated into SQL columns $(f_1, f_2, ..., f_n)$, in the right order.

The Figure 6.8 shows the `READ NEXT` paragraph for the file `STUDENT` described above. Thanks to the last statement of this paragraph (`MOVE SQLCODE TO P2-STATUS`), we can simulate the `AT END` exception. When the value of `SQLCODE` is equal to 100, it means that the SQL fetch failed, which corresponds to the COBOL *END OF FILE*. In this case, `P2-STATUS-AT-END` is set to TRUE.

6.3.6 READ KEY IS paragraphs

The random `READ` can be seen as a `START` with *equal* key usage, directly followed by a `READ NEXT`. So, the `READ KEY IS` paragraphs consist of opening the *not less* cursor and fetching it. Of course, we first test the `INVALID KEY` exception with a `SELECT COUNT` clause, as shown in Figure 6.9.

6.3.7 WRITE paragraphs

The `WRITE` paragraph is very simple. It consists of inserting a rowin the table the record was translated into, using the SQL `INSERT` query. We just have to mention in the `VALUES` clause the fields that have been translated into SQL columns $(f_1, f_2, ..., f_n)$, in the right order. In Figure 6.10, we give the example of the `WRITE` paragraph for the file `STUDENT` described above.

```
P2-READ-STUDENT-NEXT.
    IF(LAST-CURSOR-STUD = "ORDER_BY_NAME")
        EXEC SQL
          FETCH ORDER_BY_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD = "GREATER_NAME")
        EXEC SQL
          FETCH GREATER_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD = "NOT_LESS_NAME")
        EXEC SQL
          FETCH NOT_LESS_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD="ORDER_BY_LAST_NAME")
        EXEC SQL
          FETCH ORDER_BY_LAST_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD="GREATER_LAST_NAME")
        EXEC SQL
          FETCH GREATER_LAST_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD="NOT_LESS_LAST_NAME")
        EXEC SQL
          FETCH NOT_LESS_LAST_NAME
          INTO :FIRST-NAME,:LAST-NAME,:ADDRESS,:SCHOOL
        END-EXEC.
    MOVE SQLCODE TO P2-STATUS.
```

Figure 6.8: `READ NEXT` *paragraph for the file* `STUDENT`

```
P2-READ-STUDENT-KEY-NAME.
      PERFORM CLOSE-LAST-CURSOR-STUD.
      EXEC SQL
        SELECT COUNT(*)
        INTO :P2-COUNTER
        FROM STUD
        WHERE FIRST_NAME = :FIRST-NAME AND
              LAST_NAME = :LAST-NAME
      END-EXEC.
      IF (SQLCODE NOT = 0) %% SQL ERROR
         SET P2-STATUS-SQL-ERROR TO TRUE
      ELSE
         IF (P2-COUNTER = 0) %% invalid key !
            SET P2-STATUS-INVALID-KEY TO TRUE
         ELSE
            EXEC SQL
              OPEN NOT_LESS_NAME
            END-EXEC
            MOVE "NOT_LESS_NAME" TO LAST-CURSOR-STUD
            EXEC SQL
              FETCH NOT_LESS_NAME
              INTO :FIRST-NAME,
                   :LAST-NAME,
                   :ADDRESS,
                   :SCHOOL
            END-EXEC
            MOVE SQLCODE TO P2-STATUS
         END-IF
      END-IF.
```

Figure 6.9: READ KEY IS *paragraph for the file* STUDENT

```
P2-WRITE-STUD.
     EXEC SQL
       INSERT INTO STUD
       VALUES (:FIRST-NAME,
               :LAST-NAME,
               :ADDRESS,
               :SCHOOL)
     END-EXEC
     MOVE SQLCODE TO P2-STATUS.
```

Figure 6.10: WRITE *paragraph for the file* STUDENT

```
P2-REWRITE-STUD.
     EXEC SQL
       UPDATE STUD
       SET FIRST_NAME = :FIRST-NAME,
           LAST_NAME = :LAST-NAME,
           ADDRESS = :ADDRESS,
           SCHOOL = :SCHOOL
       WHERE FIRST_NAME = :FIRST-NAME AND
             LAST_NAME = :LAST-NAME
     END-EXEC.
     MOVE SQLCODE TO P2-STATUS.
```

Figure 6.11: `REWRITE` *paragraph for the file* `STUDENT`

```
P2-DELETE-STUDENT.
     EXEC SQL
       DELETE
       FROM STUD
       WHERE FIRST_NAME = :FIRST-NAME AND
             LAST_NAME = :LAST-NAME
     END-EXEC.
     MOVE SQLCODE TO P2-STATUS.
```

Figure 6.12: `DELETE` *paragraph for the file* `STUDENT`

6.3.8 REWRITE paragraphs

The `REWRITE` paragraph simply consists of a SQL update of the record (i.e., the row) with the current primary key value. This is true for both sequential and random accesses. For the sequential access, the record must be read before it can be rewritten, such that the current value of the primary key is the key value of the record to rewrite. For the random access, the programmer has first to move a value to the primary key. In Figure 6.11, we give the example of the `REWRITE` paragraph for the file `STUDENT` described above.

6.3.9 DELETE paragraphs

The `DELETE` statement can be simply translated into a SQL delete. The record (i.e., the row) to be deleted is the record having the same primary key value as the current primary key value. This is true for both sequential and random `DELETE`. For the sequential access, the record must be read before it can be deleted, such that the current value of the primary key is the key of the record to delete. For the random access, the programmer has first to move a value to the primary key.

6.4 Legacy code transformations

Remember that we clearly separate the legacy code transformation and the new SQL code generation. The reasons of doing this are quite easy to understand. First, it allows

```
READ STUDENT KEY IS NAME
INVALID KEY MOVE 1 TO END-FILE
```

a)before conversion

```
*READ STUDENT KEY IS NAME
  MOVE "read" TO SQL-ACTION
  MOVE "KEY IS NAME"
    TO SQL-OPTION
  CALL "WR-STUD" USING
    BY CONTENT SQL-ACTION
    BY REFERENCE ORD
    BY CONTENT SQL-OPTION
    BY REFERENCE WR-STATUS

*INVALID KEY
  IF WR-STATUS-INVALID-KEY
    THEN
      MOVE 1 TO END-FILE
  END-IF
```

b) access-to-call

```
*READ STUDENT KEY IS NAME
 PERFORM P2-READ-STUDENT-KEY-NAME

*INVALID KEY
 IF P2-STATUS-INVALID-KEY
   THEN
     MOVE 1 TO END-FILE
 END-IF
```

c) access-to-perform

Figure 6.13: *Comparison of P1 and P2 conversions*

us to keep a higher control on the program transformation. Secondly, it guarantees an easier maintenance of the generated code. Last, it allows us to reuse the same implementation approach as P1: "`CALL`+wrappers" becomes "`PERFORM`+paragraphs". Once all the P2 paragraphs have been generated in the program, the COBOL access statements accessing migrated files have to be replaced with the corresponding `PERFORM` statements. As we defined the P1 transformer `access-to-call`, we defined the P2 transformer `access-to-perform`. These two transformers are very similar. As a comparison, Figure 6.13 shows both `access-to-call` and `access-to-perform` effects on a COBOL random `READ` statement. The function `access-to-perform` replaces the random `READ` of the file `STUDENT` using the access key `NAME` with the `PERFORM` statement executing the paragraph `P2-READ-STUDENT-KEY-NAME` translating this random reading into SQL. This generated paragraph is given in Figure 6.9.

6.5 Optimization

Some generated P2 paragraphs may not be effectively used by the transformed COBOL program. The P2 section can be large, and only very few of its paragraphs could be effectively called. This is the reason why we wrote another accumulator function, returning the names list of the P2 paragraphs being effectively called from the legacy rewritten code. In others words, the returning list contains the names of the P2 paragraphs for which at least one `PERFORM` statement has been found. Here is the SDF definition of that accumulator:

```
p2-called-parags(Cobol-source-program, Mylist)
    -> Mylist {traversal(accu)}
```

```
[final] finaltransfo(program1, myparam, file-list1) = program5

when
 rn-table = record-name-table(program1,[],file-list1),
 rk-table = record-key-table(program1,[],file-list1),
 myparam = myparameters(rt(rt-table),
                        tc(tc-table),
                        rf(rf-table),
                        tk(tk-table)),
 table-name*1 = values2(rt-table),
 datadec1* = datalist(program1,
                      01 P2-STATUS PIC S9(9).
                        88 P2-STATUS-NO-ERR VALUE 0.
                        88 P2-STATUS-INVALID-KEY VALUE 1.
                        88 P2-STATUS-AT-END VALUE 100.
                        88 P2-STATUS-SQL-ERROR VALUE 99.
                      01 P2-COUNTER PIC X(100).
                      last-cursor-decs(table-name*1),file-list1),

 program2 = file-to-working(program1,
                      EXEC SQL INCLUDE SQLCA END-EXEC.
                      EXEC SQL BEGIN DECLARE SECTION END-EXEC.
                      datadec1*
                      EXEC SQL END DECLARE SECTION END-EXEC., file-list1),

 program3 = open-and-close-unfolder(program2),
 program4 = access-to-perform(program3,rn-table,rk-table),
 parags-list1 = p2-called-parags(program4,mylist(P2-CURSORS-DECLARATION)),
 program5 = p2-section(program4,myparam,rn-table,rk-table,parags-list1)
```

Figure 6.14: *P2 conversion equation*

6.6 P2 conversion equation

The P1 transformer `file-to-working` and the P1 accumulators `datalist`, `record-name-table` and `record-key-table` are reused in P2 transformation. The Figure 6.14 shows the top-level equation of the P2 transformation. This equation is very similar to the P1 top-level equation presented in Section 5.3.1. The differences are the following:

- The function `finaltransfo` has got an extra argument (`myparam`) consisting of the mapping needed to generate the P2 paragraphs
- We apply the function `access-to-perform` instead of the function `access-to-call`
- We construct the list of the effectively called P2 paragraphs (`parags-list1`).
- We add the P2 generated section at the end of the program, by using the function `p2-section`. We generates only the P2 paragraphs that are needed.
- The additional data declarations we add in the program are different, and are declared in a SQL `DECLARE` section

Specification / Measurement	**P1**	**P2**
top-module size (LOC)	3668	4312
SDF part (LOC)	160	381
Number of productions	21	63
Number of variables	71	122
Number of accumulators	3	4
Number of transformers	4	6
ASF part (LOC)	3508	3931
Number of equations	126	202
Parse table size	2,9MB	3,2MB
Equations table size	95,3KB	156,1KB

Table 6.1: *ASF+SDF specifications statistics*

Specification / Measurement	**P1**	**P2**
Parse table generation time	1 min 25	1 min 40
Equations table generation time	4 min 35	4 min 25
Parsing time		
385 LOC	$<$ 1 sec	1,9 sec
3128 LOC	3 sec	3,4 sec
31 000 LOC	18 sec	20 sec
213 600 LOC	2 min	2 min 10
Rewriting time		
385 LOC	1,7 sec	1,8 sec
3128 LOC	9 sec	9,5 sec
31 000 LOC	2 min 40	2 min 44
213 600 LOC	70 min 15	71 min

Table 6.2: *ASF+SDF specifications performance*

6.7 P1 vs P2 specification

Here we present some comparative statistics about our P1 and P2 ASF+SDF specifications.

Table 6.1 gives a comparison between the P1 and P2 specifications. We can see that they are quite similar in size. The P2 specification is a bit more complex than the P1 specification due to the P2 paragraphs generation. It defines more SDF productions and ASF equations.

Table 6.2 gives an idea of the performance of the ASF+SDF interpreter. Note that the performance are similar between the two specifications. In order to parse a COBOL program with `sglr`, we need the parse table of the COBOL syntax. The generation of this parse table takes around 1 min 30. To apply the transformation rules to a program with `asfe`, we need the table of the equations. Its generation takes around 4 min. We tested the transformations on our case study (presented in Chapter 7). We artificially duplicated its `PROCEDURE` division to have an idea of the parsing/rewriting time for large programs[3].

[3]These tests were made on a laptop (Processor Intel Centrino 1,3GhZ - Memory 512MB DDR)

Chapter 7

Case study

In this Chapter, we present a case study, by applying a Database First migration strategy to a small COBOL application. In Section 7.1 we present the COBOL application accessing files that we want to migrate to a SQL database. In Section 7.2, we describe the migration of the application with respect to the *Wrapper* Strategy (<D2,P1>). Section 7.3 presents the migration of the application following the *Statement Rewriting* strategy (<D1,P2>). In both cases, we first describe the database conversion step before illustrating the automatic program conversion step.

7.1 A Small Cobol Application

The case study we will present is not a real program, but was designed to illustrate some difficulties that are meet in real legacy COBOL applications. The same case study was used in [Hai02] and [Hen03]. The COBOL program, called `order.cob`, contains almost 400 lines of code and manipulates three indexed files:

- `CUSTOMER` : made up records related to the customers.
- `ORDERS` : made up records related to the orders of the customers.
- `STOCK` : made up records related to the stocks of products that can be ordered.

These files are defined in the program as shown in Figure 7.1.

7.2 Wrapper Strategy

7.2.1 D2 Database Conversion

The D2 database conversion strategy, or *Conceptual Conversion*, consists of:

1. recovering the conceptual schema of the legacy database, through a database reverse engineering phase.
2. constructing the new database from this conceptual schema, by following a standard database methodology.

The database reverse engineering of `C-ORD` is described in details in [Hen03]. We will briefly discuss and illustrate the two main steps of this DBRE process, which are the *data structure extraction* and the *data structure conceptualization*.

```
FILE-CONTROL.
 SELECT CUSTOMER ASSIGN TO "c:\CUSTOMER"
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS CUS-CODE.
 SELECT ORDERS ASSIGN TO "c:\ORDERS"
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS ORD-CODE
   ALTERNATE RECORD KEY IS ORD-CUSTOMER
     WITH DUPLICATES
   ALTERNATE RECORD KEY IS ORD-DATE
     WITH DUPLICATES.
 SELECT STOCK ASSIGN TO "c:\STOCK"
   ORGANIZATION IS INDEXED
   ACCESS MODE IS DYNAMIC
   RECORD KEY IS STK-CODE.
```

```
FILE SECTION.
FD CUSTOMER.
 01 CUS.
   02 CUS-CODE PIC X(12).
   02 CUS-DESCR PIC X(110).
   02 CUS-HIST PIC X(1000).
 FD ORDERS.
 01 ORD.
   02 ORD-CODE PIC 9(10).
   02 ORD-DATE PIC X(8).
   02 ORD-CUSTOMER PIC X(12).
   02 ORD-DETAIL PIC X(200).
 FD STOCK.
 01 STK.
   02 STK-CODE PIC 9(5).
   02 STK-NAME PIC X(100).
   02 STK-LEVEL PIC 9(5).
```

Figure 7.1: *Definition of three COBOL files*

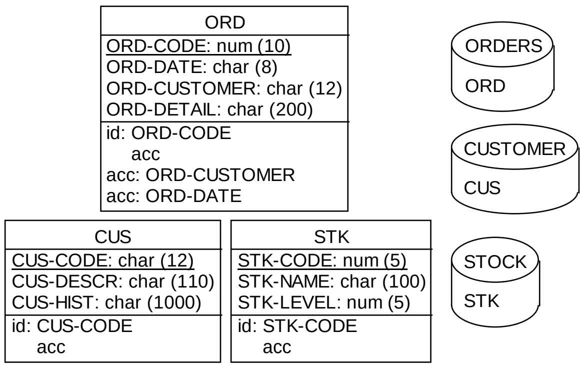


Figure 7.2: *Raw physical schema extracted from the COBOL program*

Data Structure Extraction

The data structure extraction is the most crucial and difficult part of the DBRE. Data structure extraction analyzes the existing (legacy) system to recover the complete logical schema. [Hen03]

In short, the logical schema is the schema the programmer must understand to be able to modify the legacy database and the programs that modify the data. So the logical schema includes all the constraints that must be known by the programmer, e.g., the referential constraints.

The aim of the data structure extraction is to recover this logical schema. The data structure extraction of the case study consists of the following steps:

1. *DDL code analysis*: By analyzing the DDL code shown in Figure 7.1, the raw physical schema can be extracted. As shown in Figure 7.2, this physical schema contains all the data structures declared, and only them. The data structure extraction is performed with respect to the abstraction rules given in Table 7.1 [Hai02].

COBOL statement	**Physical abstraction**
`SELECT` S `ASSIGN TO` P	collection S assigned to physical file P
`RECORD KEY IS` F	attribute F is the primary identifier and an access key
`ALTERNATE RECORD KEY IS` F	attribute F is a secondary identifier and an access key
`ALTERNATE RECORD KEY IS` F `WITH DUPLICATES`	attribute F is an access key
`FD` S`.` `01` R	entity type R within storage S
`05` F `PIC 9(`n`)`	numeric attribute F of size n, associated with its parent structure (entity type or compound attribute)
`05` F `PIC X(`n`)`	alphanumeric attribute F of size n, associated with its parent structure (entity type or compound attribute)
`05` F_1`.` `10` F_2 `...`	compound attribute F_1, with sub-attribute F_2, etc.

Table 7.1: *Main abstraction rules for COBOL file structures*

2. *Schema refinement*: By analyzing the physical schema obtained, the analyst can discover hypotheses about:
   - the fine-grained structure of entity types and attributes;
   - finding referential constraints (foreign keys);
   - finding sets behind arrays;
   - finding exact cardinalities of attributes ;
   - finding identifiers of multi-valued attributes.

   Each hypothesis is to be validated through program code and data analysis. The main techniques and tools used to discover/validate hypothesis are the program slicing, the data dependency analysis and the foreign key discovery assistant. Validated hypothesis are added to the physical schema. For instance, the analyst discovered that the attribute `ORD-CUSTOMER` of the entity type `ORD` is a reference key to the primary identifier `CUS-CODE` of the entity type `CUST`. So this constraint is added to the schema.

3. *Schema cleaning* When the refinement of the physical schema is completed, it is cleaned in order to obtain the complete logical schema, describing the programmer view of the legacy database. The resulting logical schema of the case study is given in Figure 7.3.

Data Structure Conceptualization The Data Structure Conceptualization consists of the conceptual interpretation of the logical schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimizations and DMS-dependent constructs and in interpreting them [Hen03].

In particular, the logical schema of the case study given in Figure 7.3 can be conceptualized by the conceptual schema shown in Figure 7.4. The following transformations were applied:

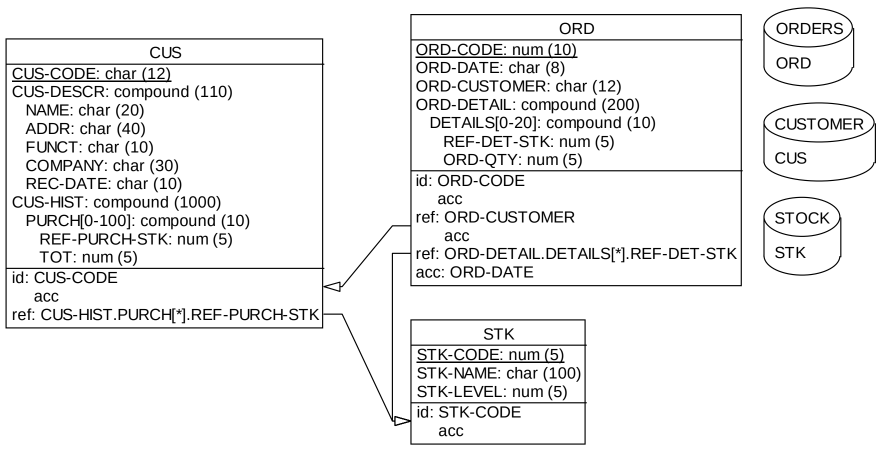


Figure 7.3: *Complete logical schema of the database*

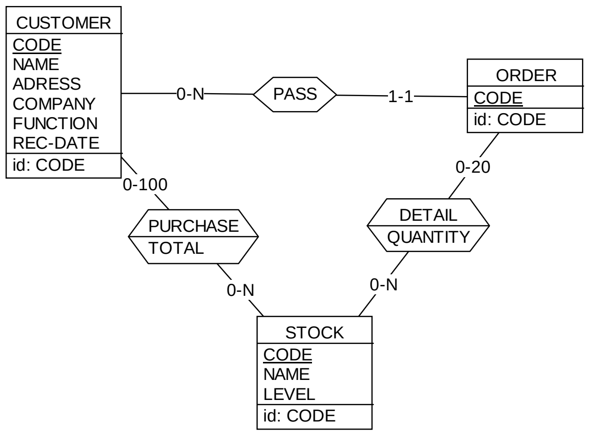


Figure 7.4: *Normalized conceptual schema of the database*

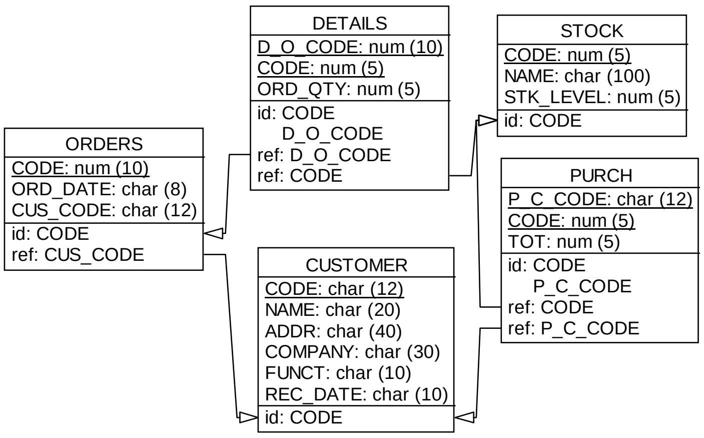


Figure 7.5: *Physical schema of the SQL database*

- The names in the schema were changed to be more meaningful. For instance, the names of the entity types (`CUS`,`ORD` and `STK`) were substituted by the names of their collection (`CUSTOMER`,`ORDER` and `STOCK`).
- The physical constructs such as the access keys and the collections were discarded.
- The complex multi-valued attributes (`PURCH` and `DETAILS`) were transformed into entity types.
- Finally, the schema was *normalized*. Through this process, the analyst tried to give the schema the qualities of a good conceptual schema such as readability, concision, minimality and expressiveness.

From the conceptual schema obtained through the DBRE phase (Figure 7.4), a SQL database can be designed. Figure 7.5 gives the physical schema of the new SQL database. Figure 7.6 gives the DDL code of the SQL database declaration.

7.2.2 P1 Program Transformation

An history log has been maintained during all the transformations that were applied from the legacy COBOL physical schema to the new SQL physical schema. This formalized history log can be seen as a mapping between the legacy physical schema and the new physical schema. This mapping is used to generate a large part of the wrappers that will encapsulate the SQL database.

In this case study, three wrappers have been generated, one for each migrated COBOL file. The naming convention we used is the following: the name of a wrapper has the form "`WR-`*record-type*", where *record-type* is the name of the record type of the migrated file. So the wrappers are:

- `WR-CUS`, for the file `CUSTOMER`;
- `WR-ORD`, for the file `ORDERS`;
- `WR-STK`, for the file `STOCK`.

```
create table CUSTOMER (
     CODE char(12) not null,
     NAME char(20) not null,
     ADDR char(40) not null,
     COMPANY char(30),
     FUNCT char(10),
     REC_DATE char(10) not null,
     primary key (CODE));

create table DETAILS (
     D_O_CODE numeric(10) not null,
     CODE numeric(5) not null,
     ORD_QTY numeric(5) not null,
     primary key (CODE, D_O_CODE));

create table ORDERS (
     CODE numeric(10) not null,
     ORD_DATE char(8) not null,
     CUS_CODE char(12) not null,
     primary key (CODE));

create table PURCH (
     P_C_CODE char(12) not null,
     CODE numeric(5) not null,
     TOT numeric(5) not null,
     primary key (CODE, P_C_CODE));
```

```
create table STOCK (
     CODE numeric(5) not null,
     NAME char(100) not null,
     STK_LEVEL numeric(5) not null,
     primary key (CODE));

alter table DETAILS add constraint FKDET_STO
     foreign key (CODE)
     references STOCK;

alter table DETAILS add constraint FKDET_ORD
     foreign key (D_O_CODE)
     references ORDERS;

alter table ORDERS add constraint FKCUSTOMER
     foreign key (CUS_CODE)
     references CUSTOMER;

alter table PURCH add constraint FKPUR_CUS
     foreign key (P_C_CODE)
     references CUSTOMER;

alter table PURCH add constraint FKPUR_STO
     foreign key (CODE)
     references STOCK;
```

Figure 7.6: *DDL code of the SQL database (D2)*

We applied our P1 rewrite rules to the program `order.cob`:

`finaltransfo`(*order.cob*, *file-list*)

The parameter of the transformation *file-list* is the list of three COBOL files that have been migrated: `mylist(CUSTOMER, ORDERS, STOCK)`

All the COBOL access statements were successfully replaced by a `CALL` to the corresponding wrapper. The `ENVIRONMENT` and `DATA` divisions were correctly reorganized. The resulting program (after pretty-printing) is available in Appendix C. This program was tested for all kind or access (read, write, rewrite,etc.). We concluded that the resulting program is equivalent to the initial `order.cob` program.

7.3 Statement Rewriting Strategy

7.3.1 D1 Database Conversion

The D1 database conversion strategy, or the *Physical conversion*, is less complex than the D2 strategy. It just translates each construct of the source database into similar constructs of the target DMS. Figure 7.7 shows the D1 conversion applied to our case study. The names are changed to comply with the SQL syntax, but the data structure does not change. Figure 7.8 gives the DDL code of the SQL database declaration.

It exists a one-to-one mapping between the source COBOL schema and the target SQL schema. This mapping is an argument of the P2 conversion of the program `order.cob`, used in the generation of the P2 paragraphs.

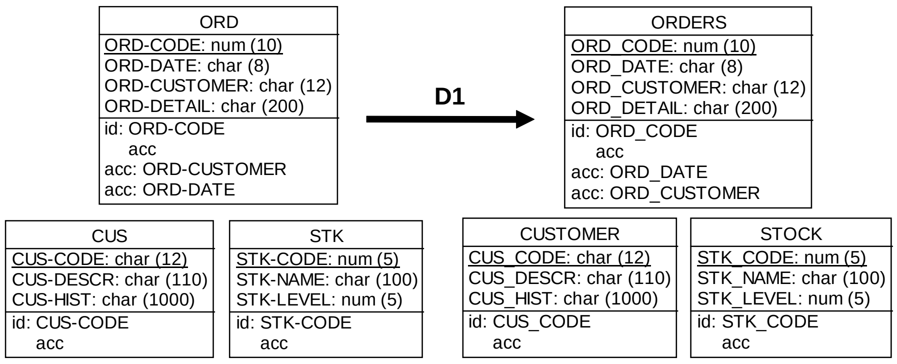


Figure 7.7: *D1 conversion of the case study*

```
create table CUSTOMER (                   create table STOCK (
     CUS_CODE char(12) not null,               STK_CODE numeric(5) not null,
     CUS_DESCR char(110) not null,             STK_NAME char(100) not null,
     CUS_HIST char(1000) not null,             STK_LEVEL numeric(5) not null,
     primary key (CUS_CODE));                  primary key (STK_CODE));

create table ORDERS (                     create index ORD_CUSTOMER
     ORD_CODE numeric(10) not null,            on ORDERS (ORD_CUSTOMER);
     ORD_DATE char(8) not null,
     ORD_CUSTOMER char(12) not null,      create index ORD_DATE
     ORD_DETAIL char(200) not null,             on ORDERS (ORD_DATE);
     primary key (ORD_CODE));
```

Figure 7.8: *DDL code of the SQL database (D1)*

```
myparameters(
    rt([<CUS,CUSTOMER>,<ORD,ORDERS>,<STK,STOCK>]),

    tc([<CUSTOMER,c(CUS_CODE CUS_DESCR CUS_HIST)>,
        <ORDERS,c(ORD_CODE ORD_DATE ORD_CUSTOMER ORD_DETAIL)>,
        <STOCK,c(STK_CODE STK_NAME STK_LEVEL)> ]),

    rf([<CUS,f(CUS-CODE CUS-DESCR CUS-HIST)>,
        <ORD,f(ORD-CODE ORD-DATE ORD-CUSTOMER ORD-DETAIL)>,
        <STK,f(STK-CODE STK-NAME STK-LEVEL)>]),

    tk([<CUSTOMER,rk([<CUS-CODE,cf(CUS_CODE,CUS-CODE,CUS_CODE)>])>,
        <ORDERS,rk([<ORD-CODE,cf(ORD_CODE,ORD-CODE,ORD_CODE)>,
                    <ORD-CUSTOMER,cf(ORD_CUSTOMER,ORD-CUSTOMER,ORD_CUS)>,
                    <ORD-DATE,cf(ORD_DATE,ORD-DATE,ORD_DATE)>])>,
        <STOCK,rk([<STK-CODE,cf(STK_CODE,STK-CODE,STK_CODE)>])>])
    )
```

Figure 7.9: *P2 conversion parameters*

7.3.2 P2 Program Transformation

We applied our P2 rewrite rules to the program `order.cob`:

`finaltransfo(`*order.cob*, *d1-mapping* , *file-list*)

The parameters of this transformation are:

- *file-list*: the list of the three COBOL files that have been migrated:

  `mylist(CUSTOMER, ORDERS, STOCK)`

- *d1-mapping*: the representation of the D1 one-to-one mapping between the Cobol physical schema and the SQL physical schema (Figure 7.9). Theses parameters consist of four tables:
  - The `rt` table gives the correspondences between COBOL records and SQL tables.
  - The `tc` table gives the columns of each SQL table.
  - The `rf` table gives the corresponding fields of each COBOL record.
  - The `tk` table gives, for each access key of each SQL table, the SQL columns composing the access key, the corresponding COBOL fields, and a name to be used in the cursor names (compatible for SQL)

The P2 section was fruitfully generated and optimized. All the COBOL access statements were successfully replaced by a `PERFORM` of the corresponding generated paragraph. The `ENVIRONMENT` and `DATA` divisions were correctly reorganized.

The resulting program (after pretty-printing) is available in Appendix D. This program was tested for all kind or access (read, write, rewrite, etc.). We concluded that the resulting program is equivalent to the initial `order.cob` program.

Measurement \ Program	`order.cob`	`p1.cob`	`p2.cob`	`p2-opt.cob`
Size (LOC)	396	607	1785	1234
Size of P2 section (LOC)	-	-	1305	754
Size of wrappers (LOC)	-	851	-	-
Number of generated paragraphs	-	-	42	21
Number of declared files	3	0	0	0
Number of `OPEN`	5	-	-	-
Number of `CLOSE`	5	-	-	-
Number of `START`	2	-	-	-
Number of `READ`	8	-	-	-
Number of `WRITE`	3	-	-	-
Number of `REWRITE`	1	-	-	-
Number of `DELETE`	2	-	-	-

Table 7.2: *Statistics of the conversion of* `order.cob`

7.4 Statistics

Table 7.2 shows the statistics of the P1 conversion and P2 conversion of the program `order.cob`. The optimized P2 conversion consists of the P2 conversion where the generated P2 section only contains the paragraphs being effectively called from the legacy rewritten code.

- `p1.cob` is the program `order.cob` transformed with respect to the Wrapper strategy.
- `p2.cob` is the program `order.cob` transformed with respect to the Statement Rewriting strategy.
- `p2-opt.cob` is the program `order.cob` transformed with respect to the Statement Rewriting strategy, where the number of generated paragraphs is minimal.

Chapter 8

Evaluation and Recommendations

In this chapter we present the global evaluation of our work consisting of automating the program conversion step of a Database First migration method. In Section 8.1, we discuss our general approach. Section 8.2 evaluates the suitability of the ASF+SDF formalisms and technologies for our data reengineering purposes. In Section 8.3 we present several other program transformation systems. Finally, recommendations for future developments or improvements are made in Section 8.4.

8.1 Approach

Our approach has been the same to automate both the Wrapper and the Statement Rewriting strategies. In both the cases, we focussed on three key factors:

- Reusability.
- Traceability.
- Complexity Isolation.

Reusability In our ASF+SDF specifications, we clearly isolate the transformation steps with respect to their specific *task*. The different tasks we identify are:

- *Code Rewriting*: replacing syntax constructs by other syntax constructs having the same semantics. For example, the functions `access-to-call` and `access-to-perform` achieve this task. Even if a `CALL` statement has not the same semantics as a `READ` statement, it allows to call a wrapper simulating the reading.
- *Code Reorganization*: moving (or removing) syntax constructs from a location in the code to another. An example is the function `file-to-working`.
- *Code Generation*: generating new code to be added into the program. This is the case of the function `p2-section` that generates an additional COBOL section and adds the latter in the program.
- *Code Analysis*: retrieving some information from the program. This is typically the role of our accumulators.

In such a way, we allow these separate transformation steps to be reused. Particularly, we were able to reuse several functions defined in the P1 specification while developing the P2 specification.

The other advantage is that we could use distinct tools to perform the separate transformation steps. For instance, using the ASF+SDF technology might not be the best way to *generate* the P2 section. Indeed, this generation is based on the mapping that exists between the legacy physical data structure and the new physical data structure. That mapping could be directly used from its origin: the DB-MAIN environment. By using a Voyager 2 program that can manipulate the mappings between the schemas, the generation could be done efficiently.

Traceability As we explained in the introduction of this thesis, the traceability of the program transformation is essential. Firstly, it improves to maintenance of the transformed application. Secondly, it improves the confidence of the people working on the initial system, since they can easier recognize the transformed program. In our specifications, the traceability consists of rewriting as comments the statements that have been transformed.

We also emphasized the way to adapt the grammar while extending a specification for another COBOL dialect. We suggested in Section 4.2 to use the Grammar Deployment Kit. The main reason is that GDK allows adapting grammar in a traceable manner.

Complexity Isolation In both program conversion strategies, the generated SQL code is isolated from the legacy code. In the Wrapper strategy, the SQL code is encapsulated within a wrapper. In the Statement Rewriting strategy, it is isolated in a separate section of the program. This complexity isolation can improve the readability of the transformed program since the alteration of the legacy code is minimal (and traceable). Consequently, this facilitates the maintenance of the migrated application.

8.2 Suitability of ASF+SDF

In this section, we discuss the merits and the limitations of the ASF+SDF formalisms and tools. In particular, we evaluate the suitability of ASF+SDF for COBOL transformations purposes.

8.2.1 Merits

Formalisms The ASF+SDF formalisms are quite easy to learn and to use. The SDF's expressiveness allows defining syntax concisely and naturally. The ASF equations in concrete syntax (i.e. user-defined syntax) are also very intuitive. The use of concrete syntax to specify transformations rules closes the conceptual distance between the way of writing programs and the way to represent them in the transformation system. The abstract transformations rules given in Appendix A are actually very close to their specification in ASF.

Traversal functions We would emphasize the essential role of the Traversal Functions built-in, provided by the ASF+SDF formalisms. As mentioned in Section 3.2.9, they facilitate the writing of a specification for both program analysis and program transformation purposes. This is a serious argument in favor of ASF+SDF. We refer to [vKV03] for more details.

Modularity ASF+SDF supports modularity. We are convinced that this modularity is essential for developing tools for a huge language like COBOL. Moreover, there are many dialects of COBOL. The modular ASF+SDF formalisms can simplify the switching from one dialect to another. For the syntax, as suggested in [vSV97], "we could store the dialect specific parts of a grammar in separate modules and simply switch from one dialect to another by adapting the import structure". For the rewriting rules, we could construct distinct top modules containing the definitions of the traversal functions and their dialect specific equations. In the scope of our project, the changes in the ASF part could be minimal. It would mainly consist of adding some optional variables in the left-hand side of our equations. But, as far as we know, the right-hand side would not change at all.

Furthermore, thanks to this modularity, it would be quite easy to define a context-free grammar for COBOL-like languages. In the same way, we could easily incorporate the CODASYL data description language (DDL), which is similar to COBOL in syntax [Joh86].

Technology The ASF+SDF formalisms are supported by the ASF+SDF Meta-Environment, discussed in Chapter 3. We would like to emphasize several merits of this interactive programming environment.

- It supports both interpretation and compilation of the ASF+SDF specifications. The compiler generates very efficient C code, that can be immediately compiled into an executable. The interpreter provides sufficient performances, and keeps the layout of sub-terms that are not rewritten.
- Several key components can be used independently of the ASF+SDF Meta-Environment, providing a high flexibility. See Section 3.3 for more details.
- The user interface, called `MetaStudio`, is user-friendly and provides multiple interactive features. We especially mention the syntax-tree visualization, that is very convenient when debugging a specification.
- It is an academic software product that can be obtained for free [1]. AS all software developed by SEN1, the Meta-Environment is released under the GPL license.

Generality ASF+SDF support the development of *arbitrary* context-free grammars, even ambiguous ones. The *generalized* `sglr` parser does not complain about conflicts in the parse table. The user of the well-known Lex+Yacc formalisms may be confronted with shift-reduce and reduce-reduce conflicts, that have to be solved manually. This can be a severe limitation when developing a specification for COBOL transformations. It would become even more dramatic when extending such a specification to deal with another COBOL dialect.

Past Applications ASF+SDF has already been used for a myriad of programming language, and for the specification of software engineering problems in diverse areas. Examples of industrial applications of ASF+SDF are presented in [vvDK$^+$96].

We particularly want to mention [Vee01], another master's thesis on COBOL automatic transformations. Its author has used the ASF+SDF Meta-Environment to restructure COBOL source code "such that the modifiability and thus maintainability is improved"[Vee01]. This restructuring process consisted of eliminating `GO-TO`'s statements and isolating new paragraphs in the source code. As case studies, very large COBOL systems were successfully transformed. This project used the useful Traversal Functions and a sub-set of the IBM-VSII COBOL grammar.

[1] Available at `http://www.cwi.nl/projects/MetaEnv`

Specification formalisms The ASF+SDF formalisms combination can also be considered as a *formal* specification language. We particularly think of the P2 paragraphs generation. We have already specified these paragraphs formally in ASF+SDF. Their generation can be implemented using some other programming language.

8.2.2 Limitations

As seen above, one one the greatest merit of the ASF+SDF formalisms is the use of concrete syntax to specify the transformation rules. But, in some cases, this could be a limitation as well. In our project, conflicts occurred between the SDF COBOL syntax and the "syntax" of the ASF equations. Let us give a short example. Remember that an ASF conditional equation contains conditions. These ASF conditions are of sort `Condition`. In the COBOL syntax, the sort `Condition` is defined as well, which can lead to surprising ambiguities while parsing the ASF equations. We could give other examples of such conflicts, which do not come from a bug of the ASF+SDF Meta-Environment. This problem is the price to pay for using concrete syntax.

The way to solve this kind of problems consists of renaming the conflicting sorts in the user-defined syntax. Once again, we recommend to use GDK in order to perform the renaming in a traceable way. Another solution could be to change the import structure of the specification, such that the sorts that interfere with the equations are hidden at the level of the equations. A GDK-functionality allowing to relocate productions in other modules will be implemented.

8.3 Related Work

In this section, we briefly present several other program transformation systems. We refer to [PTW] for further details.

Stratego/XT [Vis04] Stratego/XT is a framework for the development of transformation systems aiming to support a wide range of program transformations. This framework consists of

- the transformation language Stratego;
- the XT collection of transformations tools, providing facilities including parsing and pretty-printing.

The syntax definition formalism used in Stratego/XT is SDF. The parsing is based on the scannerless generalized LR parser `sglr`. Thus, if we want to use Stratego/XT to implement our P1 and P2 transformations, we could reuse the IBM-VSII SDF grammar.

Stratego, as ASF+SDF, is based on term rewriting. The conditional rewrite rules are of the form $Label : LHS \to RHS$ **where** s, with s a computation that should succeed in order for the rule to apply. That computation can consist of a call to a library function.

The rewrite rule can contains terms written in concrete syntax. The use of concrete syntax is indicated by quotation delimiters, e.g., the |[and]| delimiters. So if we wrote our COBOL rewrite rules in Stratego they would be quite similar to ASF equations.

The specific functionality provided by Stratego/XT is the possibility to specify (explicitly) where and in what order the rewrite rules are to be applied. This is the so-called *Rewriting Strategy*, that Stratego makes explicit and programmable. For instance, the rewriting strategy could consist of an exhaustive innermost normalization of the term, or could be a single pass top-down transformation. Stratego provides basic strategy combinators for the composition of such rewriting strategies.

Name	**Args**	**Description**
Identity		Do nothing
Fail		Raise `VisitFailure` exception
Not	v	Fail if v succeeds, and vice versa
Sequence	v_1,v_2	Do v_1, then v_2
Choice	v_1,v_2	Do v_1, if it fails, do v_2
All	v	Apply v sequentially to all immediate children until it fails
One	v	Apply v sequentially to all immediate children until it succeeds
IfThenElse	c,t,f	If c succeeds, do t, otherwise do f
Try	v	*Choice(v,Identity)*
TopDown	v	*Sequence(v,All(TopDown(v)))*
BottomUp	v	*Sequence(All(BottomUp(v)),v)*
OnceTopDown	v	*Choice(v,One(OnceTopDown(v)))*
OnceBottomUp	v	*Choice(One(OnceBottomUp(v)),v)*
AllTopDown	v	*Choice(v,All(AllTopDown(v)))*
AllBottomUp	v	*Choice(All(AllBottomUp(v)),v)*

Figure 8.1: *Overview of JJTraveler's library*

JJTraveller [vDV04] JJTraveler is a combination of a framework and a library that provide generic visitor combinators for Java. Visitor combinators are small, reusable classes that carry out specific visiting steps. They are mainly used for building program understanding tools, but they could be extended for program transformation purposes.

- The JJTraveler framework offers two generic interfaces, which are `Visitor` and `Visitable`. `Visitable` provides the minimal interface for nodes that can be visited. Visitable nodes should offer three methods:

  - `int getChildCount()`: Returns the number of child nodes.
  - `Visitable getChildAt(int i)` : Returns the i^th child node.
  - `Visitable setChildAt(int i, Visitable child)`: Modify the i^th child nodes.

  The `Visitor` interface provides a single method that takes any visitable node as argument: `Visitable visit(Visitable any)`. Each visit can succeed or fail, which can be used to control traversal behavior. Failure is indicated by a `VisitFailure` exception.

- The JJTraveler library consists of several predefined visitor combinators, relying on the generic `Visitor` and `Visitable` interfaces. Figure 8.1 gives an overview of the visitor combinators available in the library. There are two kinds of combinators: *basic* combinators and *defined* combinators. Basic combinators provide the primitive building blocks for visitor combination (*Identity*, *Fail*, *Sequence*,*Choice*, etc.). Defined combinators can be (recursively) described in terms of the basic combinators as shown in Figure 8.1. An exhaustive overview of the JJTraveler's library can be found in the online documentation[2] of JJTraveler.

[2]at `http://homepages.cwi.nl/~jvisser/doc/jjtraveler/`

JJTraveler can be used by instantiating the JJTraveler's framework and then reusing the visitor combinators in its library. The use of JJTraveler to transform a program P would consist of:

1. parsing P in order to build the abstract syntax tree (AST) of P.
2. deriving a representation of this AST such that its nodes become *visitable*. A visitable syntax tree (VST) of the program is obtained.
3. traversing (and transforming) the VST using visitor combinators
4. deriving the transformed program.

Tools exist to support the step 2: ApiGen and JJForester. The initial version of ApiGen is a tool that generates automatically implementations of abstract syntax trees in C. Then, it has been extended (by different authors) to generate AST classes for Java as well. Apigen generates the implementation of the `Visitable` interface in every generated class and some convenience classes to support generic tree traversal with JJTraveler [vMV03].

JJForester is a combined parser generator, tree builder, and visitor generator for Java. It takes language definitions in SDF as input and generates Java code that facilitates the construction, representation, and manipulation of syntax trees in an object-oriented style [KV03]. It supports visitor combinators, using JJTraveler.

TXL [Cor04] TXL is a programming language specifically designed for manipulating and experimenting with programming language notations and features using source-to-source transformation.

An example of TXL program, given in [Cor04], is shown in Figure 8.2. A TXL program typically consists of three parts:

- a context-free "base" grammar for the language to be manipulated.
- a set of context-free grammatical "overrides" (extensions or changes) to the base grammar
- a set of source transformation rules to implement transformation of the extensions to the base language.

The grammar specification uses of a notation close to BNF, with nonterminals referenced in square brackets (e.g., `[expression]`) and terminal symbols directly representing themselves.

Overrides can either completely replace the original definition of the target nonterminal, or they can refer to the previous definition using the "`...`" notation, which is read as "what it was before". These grammar redefinitions (similar to the FST tools in GDK) are the key idea that distinguishes TXL from most other program transformation tools. They increase reusability while dealing with different dialects of a programming language. They also allow for defining a suitable (sub)grammar to provide a parse more appropriate to each individual application.

TXL transformation rules specify a *pattern* to be matched, and a *replacement* to substitute for it. The nonterminal type of the pattern (e.g.,`[statement]`) is given at the beginning of the pattern, and the replacement is implicitly constrained to be of the same type.

DMS [BPM04] The DMS©[3] Software Reengineering Toolkit is a commercial program analysis and transformations system. It consists of set of tools for automating customized

[3]DMS is a Registered TradeMark of Semantic Designs Inc. Information available at `http://www.semdesigns.com/`

```
% Based on standard Pascal grammar
include "Pascal.Grm"

% Overrides to allow new statement forms
redefine
   statement
   ...
   | [reference] += [expression]
end redefine

% Transformation rule
rule main
    replace [statement]
      V [reference] += E [expression]
    by
      V:= V + (E)
end rule
```

Figure 8.2: *An example TXL program*

source program analysis, modification or translation or generation of software systems. The goal of DMS is to be able to deal with the *scale* problems of real software systems such as size and multiple languages.

Here are some of the main DMS characteristics.

- DMS also uses Generalized LR parsing. DMS based parsers take streams of lexemes and parse them according to context-free syntax. In other words, the parsing uses a lexer.
- The DMS rewrite rules are written in the DMS's Rule Specification Language. They are of the abstract form $LHS \rightarrow RHS$ **if** *condition*. The **if** condition is an optional phrase referring to the variables that occur in the LHS. That condition can consist of a call to a decision procedure, coded in PARLANSE. PARLANSE is the programming language for coding DMS.
- DMS provides a pretty-printing facility, based of constructing and composing text boxes. Prettyprinting rules can be associated with each grammar rules. The DMS prettyprinter can operate in two distinct modes, which are the "prettyprinting" and the "fidelity" mode. In prettyprinting mode, it applies the supplied prettyprinting rules. In fidelity mode, it preserves the layout of the unchanged parts of the code, and honors the pretty-printing rules while producing new code.

8.4 Recommendations

To conclude this chapter, we would like to express several recommendations for future developments in this research area. Those recommendations are justified by the three key factors we identified above: reusability, traceability and complexity isolation. These factors can be regarded at the level of the IS migration strategies as well as the tools to use to perform the different steps of the migration.

8.4.1 Generic Conversion Strategy

Our approach to implement both P1 and P2 conversion strategies allow to derive a *generic* program conversion strategy. Indeed, we implemented the Statement Rewriting strategy in such a way that it becomes very close to the Wrapper strategy. The generated P2 section can be regarded as an *intern wrapper*. As suggested above, this intern wrapper should be generated in the same way as the extern wrappers called in the Wrapper strategy. The P2 section generated in the <D1,P2> strategy basically contains the same kind of code as an extern <D2,P1> wrapper. The only difference is that the mapping between the COBOL data structure and the SQL data structure is one-to-one, since the D1 conversion only translates the data structure from COBOL to SQL. Conceptually, the two "wrappers" are identical in the sense that the rules used to generate them are the same. Thus, we recommend to consider the generation of the <D1,P2> intern wrapper as a particular case of the generation of the <D2,P1> extern wrapper.

In this way, the *generic* program conversion strategy we propose does not depend on the way the database has been migrated (D1 or D2). It would consists of the two following steps:

1. deriving an inverse wrapper from the mapping that exists between the source and the target physical schemas. The mapping can be one-to-one (D1) or not (D2). The generated wrapper (that can consists of several programs) encapsulates the new database and provides to the legacy programs a "legacy" interface for accessing the migrated data.

2. transforming the legacy programs with respect to the Wrapper strategy, consisting of replacing the DMS statements with wrapper invocations.

In other words, we recommend to choose the P1 strategy instead of the P2 strategy. The greatest merits of this solution are the following :

- *Traceability*: the alteration of the legacy code is minimal, making the transformed programs more readable, more understandable and therefore easier to maintain.

- *Reusability*: A unique wrapper can be used by all the programs of the system that access the migrated data. In the P2 strategy, the generated SQL code (i.e., the P2 paragraphs) would be duplicated.

- *Complexity isolation*: the complexity of the transformation is isolated and encapsulated within the generated wrapper.

8.4.2 Tools

We are convinced that the DB-MAIN environment is efficient to support database migration, database reverse engineering, database forward engineering and wrapper generation. We have also shown that the ASF+SDF Meta-Environment is suitable for program transformations. So we recommend to work with these technologies for further developments in data reengineering. We also recommend the use of the Grammar Deployment Kit (GDK), for generating, adapting and disambiguating the SDF syntax. Let us consider these tools with respect to our three key factors:

- *Traceability*:

  – The DB-MAIN history log keeps the trace of all the transformations that were applied during the schema conversion.

  - The ASF rewrite rules written in concrete syntax allow the reader to understand the source code transformations that where applied. Moreover, the traceability can be improved by rewriting the transformed statements as comments.
  - The GDK tools allow the grammar to be adapted in a traceable manner.

- *Reusability*:

  - The modularity of the ASF+SDF formalisms increases the reusability and it also simplifies the switching from one dialect to another. Furthermore, the different components of the ASF+SDF Meta-Environment (`sglr`, `asfe`, etc.) can be reused independently of the environment.
  - The set of DB-MAIN tools can be used in different data reengineering processes (database engineering, DBRE, program analysis, wrapper generation, etc.). Moreover, the Voyager 2 language allows a high flexibility in the use of DB-MAIN.
  - The GDK tools allow the reuse an existing grammar by adapting it to a particular application.

- *Complexity isolation*: the DB-MAIN CASE tool, the ASF+SDF Meta-Environment and GDK significantly reduce the work to be done manually while converting data-intensive applications.

8.4.3 Wrapper-based architecture

Figure 8.3 depicts the Wrapper-based architecture we propose for data reengineering. The migration starts with the database conversion process consisting of schema conversion and data instances migration. To support this first process, we recommend to use the DB-MAIN environment. Then, the program conversion step can start. From the transformations history maintained by DB-MAIN, a data wrapper can be generated. This wrapper can be either extern (recommended) either intern. Finally, the legacy programs are transformed, using an ASF+SDF specification, so that they access the new database instead of the legacy (migrated) data. The SDF grammar used to parse the programs can be adapted thanks to GDK. Note that the pre-processing and post-processing phases are ignored in that general architecture.

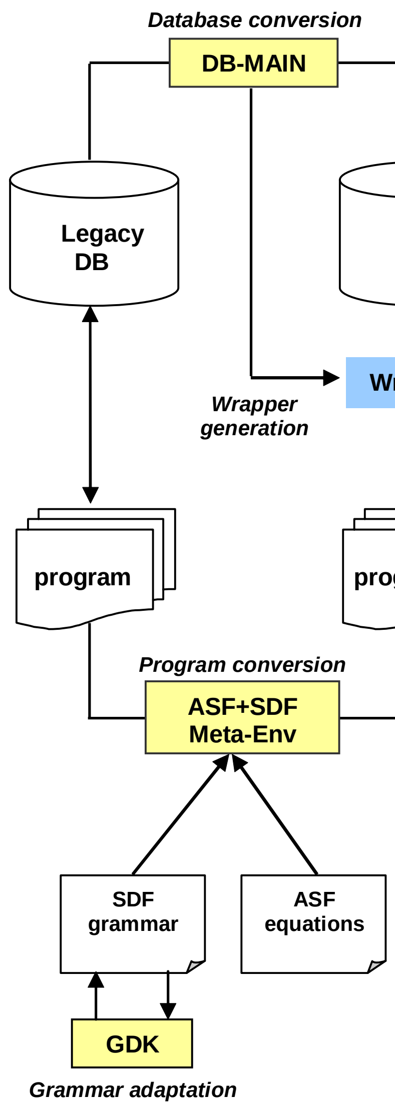


Figure 8.3: *Wrapper-based architecture for data reengineering*

Chapter 9

Conclusions

9.1 Contributions

In Chapter 1, we put the purpose of this thesis into context. We gave an overview of the Data Reengineering strategies, focusing on two of them: the <D2,P1> and <D1,P2> strategies. In Chapter 2, we presented the <D2,P1> and <D1,P2> strategies in the particular case of the conversion of COBOL files into a SQL database. We introduced the COBOL file management system and we explained how a COBOL program can be transform with respect to both P1 and P2 conversion strategies. In Chapter 3, we gave an overview of the ASF+SDF Meta-Environment that we used in our project. In Chapter 4, we proposed a general approach for COBOL programs transformation, from pre-processing to pretty-printing. In Chapter 5 we presented our ASF+SDF specification of the COBOL P1 program conversion strategy, following a D2 database conversion strategy. In Chapter 6, we presented our ASF+SDF specification implementing the COBOL P2 program conversion strategy, following a D1 (COBOL to SQL) database conversion strategy. In Chapter 7, we validated our results by applying them to a small case study. In Chapter 8, we evaluated our approach and the suitability of the tools we used. We gave some recommendations for further developments.

In conclusion, we have proved that both P1 and P2 program conversion strategies can be fully automated. More generally, we have identified three key factors for program conversion in the context of data reengineering, namely *reusability*, *traceability* and *complexity isolation*. These three key factors justify our recommendations at different levels:

- The system migration approach: an incremental database first migration method allows to maximize the reusability of the legacy data and functionalities.
- The data reengineering tools: we recommend to use DB-MAIN (database conversion - wrapper generation), the ASF+SDF Meta-Environment (legacy code transformation) and GDK (grammar adaptation).
- The program conversion strategy: the Wrapper strategy (P1) is an elegant conversion approach. It minimizes the legacy programs alteration and it encapsulates the complexity.
- The way of implementing the program conversion strategies: we recommend to clearly separate *code rewriting* and *code generation*.

9.2 Future directions

We are aware that the tools we constructed are only first prototypes. Much improvements are to be done to transform these prototypes into usable results. However, our work can be seen as a starting point for further research developments. We anticipate three possible future directions.

- The <D2,P1> and <D1,P2> strategies and tools have to be improved and validated with "real-world" COBOL systems. In particular, our two ASF+SDF specifications should be duplicated and adapted, in order to transform programs written in other COBOL dialects.
- The automation of the other COBOL data reengineering strategies could be explored. For instance, the <D2,P3> strategy produces a high quality renovated database and transforms the legacy programs in order to to use the full power of the new DMS. So, this expensive migration strategy can be justified if the whole system (database and programs) have to be renovated for the long term.

  The *Logic Rewriting* (P3) program conversion strategy requires a deep understanding of the program logic, since the latter will generally be changed. The complexity of the P3 transformation prevents its complete automation. However, program understanding tools could be used in order to support the Logic Rewriting. Their goals would be to detect *patterns* of statements that should be replaced and to give hints on how to rewrite them. In this way, the manual work could be significantly reduced.
- The data reengineering tools could be extended for other data structures and languages. For instance, the CODASYL Data Description Language (DDL), used in many COBOL programs, is much more complex than the COBOL DDL. CODASYL data reengineering seems to be another important challenge, especially at the level of wrapper generation.

Acronyms

ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
CASE	Computed Aided Software Engineering
COBOL	Common Business Oriented Language
CODASYL	Conference on Data Systems Languages
CS	Conceptual Schema
DBMS	DataBase Management System
DBRE	DataBase Reverse Engineering
DDL	Data Description Language
DML	Data Management Language
DMS	Data Management System
DMS©	a Registered TradeMark of Semantic Designs Inc
GER	Generic Entity-Relationship
GDK	Grammar Deployment Kit
IMS	Information Management System, non-relational Data Base System
SDF	Syntax Definition Formalism
SPS	Source Physical Schema
TPS	Target Physical Schema
V2	Voyager 2

Bibliography

[Ben95] K.H. Bennett. Legacy Systems: Coping With Success. In *IEEE Software*, volume 12, pages 19–23, 1995.

[BPM04] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS©: Program Transformations for Practical Scalable Software Evolution. In *Proceedings of International Conference of Software Engineering*, 2004. `http://www.cwi.nl/events/2002/GP2002/papers/baxter.pdf`.

[Bro98] G. DeWard Brown. *Advanced Cobol For Structured and Object Oriented Programming*. John Wiley, third edition, 1998.

[Bru03] Magiel Bruntink. Testability of Object-Oriented Systems: a Metrics-based approach. Master's thesis, Faculty of Natural Sciences, Mathematics, and Computer Science, University of Amsterdam, September 2003.

[BS95] M.L. Brodie and M. Stonebraker. *Migrating Legacy Systems. Gateways, Interfaces and the incremental approach*. Morgan Kaufmann Publishers, 1995.

[Cla81] Andre Clarinval. *Comprendre, Connaître et Maîtriser le Cobol*. Presses Universitaires de Namur, second edition, 1981.

[Cor04] J.R. Cordy. TXL - A Language for Programming Language Tools and Applications. In *Proc. LDTA 2004, ACM 4th International Workshop on Language Descriptions, Tools and Applications*, pages 1–27, Barcelona, April 2004.

[DBM] The DB-MAIN official website. `http://www.db-main.be`, Last accessed 18 May 2004.

[Hai00] Jean-Luc Hainaut. *Bases de Données et Modèles de Calcul, Outils et Methodes pour l'utilisateur*. Dunod, second edition, 2000.

[Hai02] Jean-Luc Hainaut. Introduction to Database Reverse Engineering. at `http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf`, 2002.

[Hen03] Jean Henrard. *Program Understanding in Database Reverse Engineering*. PhD thesis, University of Namur, 2003.

[HHTH02] Jean Henrard, Jean-Marc Hick, Philippe Thiran, and Jean-Luc Hainaut. Strategies for Data Reengineering. In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 211–220. IEEE Computer Society Press, 2002.

[Joh86] L.F. Johnson. *File Techniques for Data Base Organization in Cobol*. Prentice-Hall International, second edition, 1986.

[KLV02] Jan Kort, Ralf Lämmel, and Chris Verhoef. The Grammar Deployment Kit. In Mark van den Brand and Ralf Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[Kor03] Jan Kort. *Grammar Deployment Kit Reference Manual*, May 2003.

[KV03] T. Kuipers and J. Visser. Object-oriented tree traversal with jjforester. *Science of Computer Programming*, 47:59, April 2003.

[Lä01] Ralf Lämmel. Grammar Adaptation. In *Proc. Formal Methods Europe (FME) 2001*, volume 2021 of *LNCS*, pages 550–570. Springer-Verlag, 2001.

[LV01] Ralf Lämmel and Chris Verhoef. Semi-Automatic Grammar Recovery. *Software Practice & Experience*, 31(15):1395–1438, December 2001.

[PK81] A.S. Philippakis and Leonard J. Kazmier. *Structured Cobol.* McGraw-Hill, second edition, 1981.

[PK82] A.S. Philippakis and Leonard J. Kazmier. *Advanced Cobol.* McGraw-Hill, 1982.

[PTW] The TWiki website dedicated to Program Transformation. `http://www.program-transformation.org`, Last modified 06 May 2004, Last accessed 8 May 2004.

[SPL03] Robert C. Seacord, Daniel Plakosh, and Grace A. Lewis. *Modernizing Legacy Systems.* SEI Series in Software Engineering. Addison-Wesley, 2003.

[TH01] Philippe Thiran and Jean-Luc Hainaut. Wrapper Development for Legacy Data Reuse. In *Proc. of the 8th Working Conference on Reverse Engineering (WCRE'01).* IEEE Computer Society Press, 2001.

[Ulr02] William M. Ulrich. *Legacy Systems: Transformation Strategies.* Prentice Hall PTR, 2002.

[vDV04] A. van Deursen and J. Visser. Source Model Analysis Using the JJTraveler Visitor Combinator Framework. *Software: Practice and Experience*, 2004.

[Vee01] Niels Veerman. Restructuring Cobol Systems using Automatic Transformations. Master's thesis, Vrije Universiteit Amsterdam, 2001.

[Vis04] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer et al., editors, *Domain-Specific Program Generation*, Lecture Notes in Computer Science. Spinger-Verlag, June 2004. (To appear).

[vK03] Mark G.J. van den Brand and Paul Klint. *ASF+SDF Meta-Environment User Manual*, September 2003.

[vKV03] Mark G.J. van den Brand, Paul Klint, and Jurgen J. Vinju. Term Rewriting with Traversal Functions. *ACM Transactions on Software Engineering Methodoly*, 12(2):152–190, 2003.

[vMV03] Mark G.J. van den Brand, P.E. Moreau, and J.J. Vinju. A generator of efficient strongly typed abstract syntax trees in java. Technical Report SEN-E0306, CWI, 2003. available at `http://www.cwi.nl/themes/sen1/twiki/pub/SEN1/ApiGen/submitted-20-11-200%3.pdf`.

[vSV97] Mark G.J. van den Brand, Alex Sellink, and Chris Verhoef. Obtaining a COBOL Grammar from Legacy Code for Reengineering Purposes. In *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)*, 1997.

[vV00] Mark G.J. van den Brand and J.J. Vinju. Rewriting with Layout. In Claude Kirchner and Nachum Dershowitz, editors, *Proceedings of RULE2000*, 2000.

[vvDK$^+$96] Mark G. J. van den Brand, Arie van Deursen, Paul Klint, Steven Klusener, and E. A. van der Meulen. Industrial Applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST'96)*, volume 1101 of *Lecture Notes in Computer Science*, pages 9–18. Springer-Verlag, 1996.

[Wie95] G. Wiederhold. Modeling and System Maintenance. In *Proceedings of the International Conference on Object-Oriented and Entity-Relationship Modeling*, 1995.

[Wik] the Free Encyclopedia Wikipedia. at `http://en.wikipedia.org`, Last modified 22 Mar 2004, Last accessed 06 Apr 2004.

[WLB$^+$97] B. Wu, D. Lawless, J. Bisbal, J. Grimson, V. Wad, D. O'Sullivan, and R. Richardson. Legacy System Migration: A Legacy Data Migration Engine. In Ed. Czechoslovak Computer Experts, editor, *Proccedings of the 17th International Database Conference (DATASEM '97)*, pages 129–138, 1997.

Appendix A

Abstract Transformation rules

In this appendix, we will give a summary of *some* transformation rules, used to replace the COBOL DMS statements according to both <D2,P1> and <D1,P2> strategies. In both cases, these rules allow the program to access the SQL tables instead of the migrated COBOL files. Remember that in the <D2,P1> strategy, we locally replace the COBOL DMS statements with a `CALL` statement to a generated wrapper. In the <D1,P2> strategy, we locally replace the COBOL DMS statements with a `PERFORM` statement executing a generated paragraph.

These rules are not exhaustive, we give *one* rule for each kind of COBOL access statement (`OPEN`, `CLOSE`, `START`, sequential `READ`, random `READ`, etc.).

OPEN

Syntax

```
OPEN open-option file-name
```

<D2,P1>

```
  CALL wrapper-name
    USING
           “open ”
           record-name
           open-option
           wrapper-status
```

<D1,P2>

```
  PERFORM
    P2-OPEN-open-option-file-name
```

CLOSE

Syntax

```
  CLOSE file-name
```

<D2,P1>

```
  CALL wrapper-name
    USING
           “close ”
           record-name
           no-option
           wrapper-status
```

<D1,P2>

No translation is necessary.

START

Syntax

```
  START file-name KEY IS operator access-key
       INVALID KEY statements-1
       NOT INVALID KEY statements-2
  END-START
```

<D2,P1>

```
  CALL wrapper-name
    USING “start ”
          record-name
          “KEY IS operator access-key ”
          wrapper-status
  IF  wrapper-status-invalid-key
    THEN  statements-1
    ELSE  statements-2
  END-IF
```

<D1,P2>

```
  PERFORM
    P2-START-file-name-operator-access-key
  IF  p2-status-invalid-key
    THEN  statements-1
    ELSE  statements-2
  END-IF
```

sequential READ

Syntax

```
READ file-name NEXT
    AT END statements-1
    NOT AT END statements-2
END-READ
```

<D2,P1>

```
CALL wrapper-name
  USING "read "
        record-name
        "no-option "
        wrapper-status
IF  wrapper-status-at-end
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-READ-file-name-NEXT
IF  p2-status-at-end
  THEN  statements-1
  ELSE  statements-2
END-IF
```

random READ

Syntax

```
READ file-name KEY IS access-key
    INVALID KEY statements-1
    NOT INVALID KEY statements-2
END-READ
```

<D2,P1>

```
CALL wrapper-name
  USING "read "
        record-name
        "KEY IS access-key "
        wrapper-status
IF  wrapper-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-READ-file-name-KEY-IS-access-key
IF  p2-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

sequential WRITE

Syntax

```
WRITE record-name
```

<D2,P1>

```
CALL wrapper-name
  USING "write "
        record-name
        no-option
        wrapper-status
```

<D1,P2>

```
PERFORM P2-WRITE-record-name
```

random WRITE

Syntax

```
WRITE record-name
    INVALID KEY statements-1
    NOT INVALID KEY statements-2
END-WRITE
```

<D2,P1>

```
CALL wrapper-name
  USING "write "
        record-name
        no-option
        wrapper-status
IF  wrapper-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-WRITE-record-name
IF  p2-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

sequential REWRITE

Syntax

```
REWRITE record-name
```

<D2,P1>

```
CALL wrapper-name
  USING “rewrite ”
        record-name
        no-option
        wrapper-status
```

<D1,P2>

```
PERFORM P2-REWRITE-record-name
```

random REWRITE

Syntax

```
REWRITE record-name
    INVALID KEY statements-1
    NOT INVALID KEY statements-2
END-WRITE
```

<D2,P1>

```
CALL wrapper-name
  USING “rewrite ”
        record-name
        no-option
        wrapper-status
IF  wrapper-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-REWRITE-record-name
IF  p2-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

sequential DELETE

Syntax

```
DELETE file-name
```

<D2,P1>

```
CALL wrapper-name
  USING "delete "
         record-name
         no-option
         wrapper-status
```

<D1,P2>

```
PERFORM P2-DELETE-file-name
```

random DELETE

Syntax

```
DELETE file-name
    INVALID KEY statements-1
    NOT INVALID KEY statements-2
END-DELETE
```

<D2,P1>

```
CALL wrapper-name
  USING "delete "
         record-name
         no-option
         wrapper-status
IF  wrapper-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

<D1,P2>

```
PERFORM P2-REWRITE-record-name
IF  p2-status-invalid-key
  THEN  statements-1
  ELSE  statements-2
END-IF
```

Appendix B

User Guide

In this appendix, we describe the way to use our COBOL program transformation tools.

Requirements

- The Meta-Environment must be installed[1]. It compiles and runs on most Unix platforms.

  Note that we used an older version of the Meta-Environment than the current one. The last release (Meta-Environment 1.5) contains many radical improvements of many aspects of ASF, SDF and the Meta-Environment. There are new features, some old features have been replaced by new features, and some features have become deprecated without replacement. Old ASF+SDF specifications (like our specifications) can be upgraded automatically using the Upgrade menus in the environment.
- The use of the Perl scripts required a Perl interpreter.

CD-ROM

The CD-ROM available with this Master's thesis contains:

- The full P1 ASF+SDF Specification
- The full P2 ASF+SDF Specification
- The case study order.cob

The content of the CD-ROM is also available at `www.info.fundp.ac.be/~acleve/CD-ROM`

P1 Specification

Files The directory "`P1-Specification`" of the CD-ROM contains :

- The subdirectory "`IBM-VSII SDF`", containing the full Cobol IBM-VSII SDF grammar.
- The subdirectory "`Traversal-functions`", containing both SDF syntax rules and ASF equations of the top-module.

[1]See `http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment` for more details about the installation requirements

- The subdirectory "`Case study`", containing :
  - the case study program: "`order.cob`";
  - the transformation parameters: "`file-list.txt`";
  - the resulting program: "`orderp1.cob`".
- The subdirectory "`P1-conversion script`", containing:
  - the shell script used to perform the P1 transformation: "`p1-conversion`";
  - the parse tables: "`cobol_ibmvs2.trm.tbl`" and "`Traversal-functions.trm.tbl`";
  - the equations table: "`Traversal-functions.eqs`;
  - the Perl scripts.

P1 conversion In order to apply our P1 transformation rules to a cobol program *input.cob*, the shell script "`p1-conversion`" can be used. Make sure that `p1-conversion` is executable:

```
chmod +x p1-conversion
```

Then, a COBOL program *input.cob* can be transformed using the following command:

`./p1-conversion` *input.cob files-list output.cob*

where

- the file *files-list* contains the list of the migrated files. For instance, the file list used for the conversion of the case study is `mylist(ORDERS,CUSTOMER,STOCK)`.
- *output.cob* is the name of the resulting COBOL program.

P2 Specification

Files The directory "`P2-Specification`" of the CD-ROM contains :

- The subdirectory "`IBM-VSII SDF`", containing the full Cobol IBM-VSII SDF grammar.
- The subdirectory "`Traversal-functions`", containing both SDF syntax rules and ASF equations of the top-module.
- The subdirectory "`Case study`", containing :
  - the case study program: "`order.cob`";
  - the transformation parameters: "`file-list.txt` and "`myparameters.trm`";
  - the resulting program: "`orderp2.cob`".
- The subdirectory "`P2-conversion script`", containing:
  - the shell script used to perform the P2 transformation: "`p2-conversion`";
  - the parse tables: "`cobol_ibmvs2.trm.tbl`" and "`Traversal-functions.trm.tbl`";
  - the equations table: "`Traversal-functions.eqs`;
  - the Perl scripts.

P2 conversion In order to apply our P1 transformation rules to a cobol program *input.cob*, the shell script `"p2-conversion"` can be used. Make sure that `p2-conversion` is executable:

```
chmod +x p2-conversion
```

Then, a COBOL program *input.cob* can be transformed using the following command:

`./p2-conversion` *input.cob d1-mapping files-list output.cob*

where

- the file *d1-mapping* contains the D1 one-to-one mapping that exists between the COBOL physical schema and the SQL physical schema. This mapping is of the following form:

  `myparameters(`*rt-table*`,` *tc-table*`,` *rf-table*`,` *tk-table*`)`

  where

  – The *rt-table* gives the correspondences between COBOL records and SQL tables. It has the following form:

    `rt([<`$record_1$`,`$table_1$`>,<`$record_2$`,`$table_2$`>, ... ,<`$record_n$`,`$table_n$`>])`

  – The *tc-table* gives the columns of each SQL table:

    `tc([<`$table_1$`,c(`col_{11} ... col_{1m}`)>, ... ,<`$table_n$`,c(`col_{n1} ... col_{nl}`)>])`

  – The *rf-table* gives the corresponding fields of each COBOL record:

    `rf([<`$record_1$`,f(`$field_{11}$ ... $field_{1m}$`)>, ... ,<`$record_n$`,f(`$field_{n1}$ ... $field_{nl}$`)>])`

    Note that each SQL column (col_{ij}) occurring in the *tc-table* is the D1 translation of a COBOL field ($field_{ij}$) occurring in the *rf-table*.

  – The *tk-table* gives, for each access key of each SQL table, the SQL columns (c_{ij}) composing the access key, the corresponding COBOL fields (f_{ij}), and a name to be used in the cursor names (compatible for SQL). The *tk-table* has the following form:

    `tk([<`$table_1$`,`$rk\text{-}table_1$`>, ... ,<`$table_n$`,`$rk\text{-}table_n$`>])`

    with *rk-table* of the form:

    `rk([<`$access\text{-}key_1$`, cf(`c_{11} ... c_{1k}`,` f_{11} ... f_{1k}`,` $cursor\text{-}name_1$`)>,`
    ...
    `<`$access\text{-}key_p$`, cf(`c_{p1} ... c_{pk}`,` f_{p1} ... f_{pk}`,` $cursor\text{-}name_p$`)>])`

  Figure B.2 gives the example of the *d1-mapping* used in the case study `order.cob` discussed in chapter 7. Figure B.1 depicts the translation of the three COBOL files into three SQL tables.

- the file *files-list* contains the list of the migrated files. For instance, the file list used for the conversion of the case study is `mylist(ORDERS,CUSTOMER,STOCK)`.

- *output.cob* is the name of the resulting COBOL program.

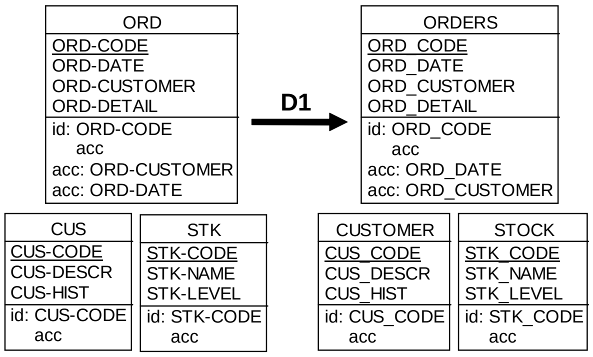


Figure B.1: *Example of D1 conversion*

```
myparameters(
    rt([<CUS,CUSTOMER>,<ORD,ORDERS>,<STK,STOCK>]),

    tc([<CUSTOMER,c(CUS_CODE CUS_DESCR CUS_HIST)>,
        <ORDERS,c(ORD_CODE ORD_DATE ORD_CUSTOMER ORD_DETAIL)>,
        <STOCK,c(STK_CODE STK_NAME STK_LEVEL)> ]),

    rf([<CUS,f(CUS-CODE CUS-DESCR CUS-HIST)>,
        <ORD,f(ORD-CODE ORD-DATE ORD-CUSTOMER ORD-DETAIL)>,
        <STK,f(STK-CODE STK-NAME STK-LEVEL)>]),

    tk([<CUSTOMER,rk([<CUS-CODE,cf(CUS_CODE,CUS-CODE,CUS_CODE)>])>,
        <ORDERS,rk([<ORD-CODE,cf(ORD_CODE,ORD-CODE,ORD_CODE)>,
                    <ORD-CUSTOMER,cf(ORD_CUSTOMER,ORD-CUSTOMER,ORD_CUS)>,
                    <ORD-DATE,cf(ORD_DATE,ORD-DATE,ORD_DATE)>])>,
        <STOCK,rk([<STK-CODE,cf(STK_CODE,STK-CODE,STK_CODE)>])>])
    )
```

Figure B.2: *Example of d1-mapping*

Appendix C

Wrapper-based Conversion of the Case Study

This appendix contains the COBOL program obtained by transforming the case study program `order.cob` with respect to the Wrapper conversion strategy. The source code fragments that have been transformed begin by a COBOL comment indicating the rewritten statement.

Here is the naming convention we used for the names of the wrappers:

- `WR-CUS`: the wrapper for the file `CUSTOMER`
- `WR-ORD`: the wrapper for the file `ORDERS`
- `WR-STK`: the wrapper for the file `STOCK`

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C-ORD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.
01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-DATE PIC X(8).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).

01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(110).
  02 CUS-HIST PIC X(1000).

01 WR-STATUS PIC 9(3).
  88 WR-STATUS-NO-ERR VALUE 0.
  88 WR-STATUS-INVALID-KEY VALUE 1.
  88 WR-STATUS-AT-END VALUE 100.

01 SQL-ACTION PIC X(100).

01 SQL-OPTION PIC X(100).

01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDR PIC X(40).
  02 COMPANY PIC X(30).
  02 FUNCT PIC X(10).
  02 REC-DATE PIC X(10).

01 LIST-PURCHASE.
  02 PURCH OCCURS 100 TIMES
      INDEXED BY IND.
    03 REF-PURCH-STK PIC 9(5).
    03 TOT PIC 9(5).

01 LIST-DETAIL.
  02 DETAILS OCCURS 20 TIMES
      INDEXED BY IND-DET.
    03 REF-DET-STK PIC 9(5).
    03 ORD-QTY PIC 9(5).

01 ORDER-DATE.
  02 ORDER-YYYY PIC X(4).
  02 ORDER-MM PIC X(2).
  02 ORDER-DD PIC X(2).
01 CHOICE PIC X.
01 END-FILE PIC 9.
01 END-DETAIL PIC 9.
01 EXIST-PROD PIC 9.
```

```
 01 PROD-CODE PIC 9(5).

 01 TOT-COMP PIC 9(5) COMP.
 01 QTY PIC 9(5) COMP.
 01 NEXT-DET PIC 99.

 PROCEDURE DIVISION.
 MAIN.
     PERFORM INIT.
     PERFORM PROCESS UNTIL CHOICE = 0.
     PERFORM CLOSING.
     STOP RUN.

 INIT.
*       OPEN I-O CUSTOMER
     MOVE "open" TO SQL-ACTION
     MOVE "i-o" TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
*       OPEN I-O ORDERS
     MOVE "open" TO SQL-ACTION
     MOVE "i-o" TO SQL-OPTION
     CALL "WR-ORD" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE ORD
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
*       OPEN I-O STOCK
     MOVE "open" TO SQL-ACTION
     MOVE "i-o" TO SQL-OPTION
     CALL "WR-STK" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE STK
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.

 PROCESS.
     DISPLAY "1 NEW CUSTOMER".
     DISPLAY "2 NEW STOCK".
     DISPLAY "3 NEW ORDER".
     DISPLAY "4 LIST OF CUSTOMERS".
     DISPLAY "5 LIST OF STOCKS".
     DISPLAY "6 LIST OF ORDERS (BY NUMBER)".
     DISPLAY "7 LIST OF ORDERS (BY DATE)".
     DISPLAY "8 DELETE CUSTOMER".
     DISPLAY "0 END".
     ACCEPT CHOICE.
     IF CHOICE = 1
       PERFORM NEW-CUS.
     IF CHOICE = 2
       PERFORM NEW-STK.
     IF CHOICE = 3
       PERFORM NEW-ORD.
     IF CHOICE = 4
       PERFORM LIST-CUS.
     IF CHOICE = 5
       PERFORM LIST-STK.
     IF CHOICE = 6
       PERFORM LIST-ORD.
     IF CHOICE = 7
       PERFORM LIST-ORD-DATE.
     IF CHOICE = 8
       PERFORM DELETE-CUS.
 CLOSING.
*      CLOSE CUSTOMER
     MOVE "close" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
*      CLOSE ORDERS
     MOVE "close" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-ORD" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE ORD
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
*      CLOSE STOCK
     MOVE "close" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-STK" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE STK
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.

 NEW-CUS.
     DISPLAY "NEW CUSTOMER :".
     DISPLAY "CUSTOMER CODE ?"
       WITH NO ADVANCING.
     ACCEPT CUS-CODE.

     DISPLAY "NAME DU CUSTOMER : "
       WITH NO ADVANCING.
     ACCEPT NAME.
     DISPLAY "ADDRESS OF CUSTOMER : "
       WITH NO ADVANCING.
     ACCEPT ADDR.
     DISPLAY "COMPANY OF CUSTOMER : "
       WITH NO ADVANCING.
     ACCEPT COMPANY.
     IF(COMPANY NOT= SPACE)
      DISPLAY "FUNCTION OF CUSTOMER : "
         WITH NO ADVANCING
       ACCEPT FUNCT
     ELSE
       MOVE SPACE TO FUNCT.
     DISPLAY "DATE : "
         WITH NO ADVANCING.
     ACCEPT REC-DATE.
     MOVE DESCRIPTION TO CUS-DESCR.
     PERFORM INIT-HIST.
*      WRITE CUS
     MOVE "write" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS
*     *INVALID KEY
     IF WR-STATUS-INVALID-KEY
        DISPLAY "ERROR"
     END-IF.

 LIST-CUS.
     DISPLAY "LISTE DES CUSTOMERS".
*      CLOSE CUSTOMER
     MOVE "close" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
*      OPEN I-O CUSTOMER
     MOVE "open" TO SQL-ACTION
     MOVE "i-o" TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS.
     MOVE 1 TO END-FILE.
     PERFORM READ-CUS
         UNTIL (END-FILE = 0).

 READ-CUS.
*      READ CUSTOMER NEXT
     MOVE "read" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-CUS" USING
       BY CONTENT SQL-ACTION
       BY REFERENCE CUS
       BY CONTENT SQL-OPTION
       BY REFERENCE WR-STATUS
*     *AT END
     IF WR-STATUS-AT-END
        MOVE 0 TO END-FILE
*         *NOT AT END
     ELSE
     DISPLAY CUS-CODE
     DISPLAY CUS-DESCR
     DISPLAY CUS-HIST
     END-IF.

 DELETE-CUS.
     MOVE 1 TO END-FILE.
     PERFORM READ-CUS-CODE
         UNTIL END-FILE = 0.
     PERFORM DELETE-CUS-ORD.
*  DELETE CUSTOMER
     MOVE "delete" TO SQL-ACTION
     MOVE " " TO SQL-OPTION
     CALL "WR-CUS" USING
     BY CONTENT SQL-ACTION
     BY REFERENCE CUS
     BY CONTENT SQL-OPTION
     BY REFERENCE WR-STATUS.
```

```
  DELETE-CUS-ORD.
      MOVE CUS-CODE TO ORD-CUSTOMER.
      MOVE 0 TO END-FILE.
*   READ ORDERS KEY IS ORD-CUSTOMER
      MOVE "read" TO SQL-ACTION
      MOVE "KEY IS ORD-CUSTOMER" TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
*  *INVALID KEY
      IF WR-STATUS-INVALID-KEY
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM DELETE-ORDER
          UNTIL END-FILE = 1.
  DELETE-ORDER.
*   DELETE ORDERS
      MOVE "delete" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS.
*   READ ORDERS NEXT
      MOVE "read" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
*  *AT END
      IF WR-STATUS-AT-END
      MOVE 1 TO END-FILE
*      *NOT AT END
      ELSE
      IF ORD-CUSTOMER NOT = CUS-CODE
      MOVE 1 TO END-FILE
      END-IF.

  NEW-STK.
      DISPLAY "NEW STOCK".
      DISPLAY "PRODUCT NUMBER : "
        WITH NO ADVANCING.
      ACCEPT STK-CODE.

      DISPLAY "NAME : "
          WITH NO ADVANCING.
      ACCEPT STK-NAME.

      DISPLAY "LEVEL : "
          WITH NO ADVANCING.
      ACCEPT STK-LEVEL.

*   WRITE STK
      MOVE "write" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-STK" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE STK
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
*  *INVALID KEY
      IF WR-STATUS-INVALID-KEY
      DISPLAY "ERREUR "
      END-IF.

  LIST-STK.
      DISPLAY "LIST OF STOCKS ".

*   CLOSE STOCK
      MOVE "close" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-STK" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE STK
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS.
*   OPEN I-O STOCK
      MOVE "open" TO SQL-ACTION
      MOVE "i-o" TO SQL-OPTION
      CALL "WR-STK" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE STK
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS.

      MOVE 1 TO END-FILE.
      PERFORM READ-STK
          UNTIL END-FILE = 0.

  READ-STK.
*   READ STOCK NEXT
      MOVE "read" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-STK" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE STK
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
*  *AT END
      IF WR-STATUS-AT-END
      MOVE 0 TO END-FILE
*      *NOT AT END
      ELSE
      DISPLAY STK-CODE
      DISPLAY STK-NAME
      DISPLAY STK-LEVEL
      END-IF.

  NEW-ORD.
      DISPLAY "NEW ORDER".
      DISPLAY "ORDER NUMBER : "
        WITH NO ADVANCING.
      ACCEPT ORD-CODE.

      DISPLAY "ORDER DATE".
      DISPLAY " DAY "
          WITH NO ADVANCING.
      ACCEPT ORDER-DD.
      DISPLAY " MONTH "
```

```
            WITH NO ADVANCING.
        ACCEPT ORDER-MM.
        DISPLAY " YEAR (YYYY) "
            WITH NO ADVANCING.
        ACCEPT ORDER-YYYY.
        MOVE ORDER-DATE TO ORD-DATE.
        MOVE 1 TO END-FILE.
        PERFORM READ-CUS-CODE
            UNTIL END-FILE = 0.
        MOVE CUS-DESCR TO DESCRIPTION.
        DISPLAY NAME.
        MOVE CUS-CODE TO ORD-CUSTOMER.
        MOVE CUS-HIST TO LIST-PURCHASE.

        SET IND-DET TO 1.
        MOVE 1 TO END-FILE.
        PERFORM READ-DETAIL
          UNTIL END-FILE = 0 OR IND-DET = 21.
        MOVE LIST-DETAIL TO ORD-DETAIL.

* WRITE ORD
        MOVE "write" TO SQL-ACTION
        MOVE " " TO SQL-OPTION
        CALL "WR-ORD" USING
        BY CONTENT SQL-ACTION
        BY REFERENCE ORD
        BY CONTENT SQL-OPTION
        BY REFERENCE WR-STATUS
**INVALID KEY
        IF WR-STATUS-INVALID-KEY
        DISPLAY "ERROR"
        END-IF.

        MOVE LIST-PURCHASE
          TO CUS-HIST.
* REWRITE CUS
        MOVE "rewrite" TO SQL-ACTION
        MOVE " " TO SQL-OPTION
        CALL "WR-CUS" USING
        BY CONTENT SQL-ACTION
        BY REFERENCE CUS
        BY CONTENT SQL-OPTION
        BY REFERENCE WR-STATUS
**INVALID KEY
        IF WR-STATUS-INVALID-KEY
        DISPLAY "ERROR CUS"
        END-IF.

  READ-CUS-CODE.
        DISPLAY "CUSTOMER NUMBER : "
          WITH NO ADVANCING.
        ACCEPT CUS-CODE.
        MOVE 0 TO END-FILE.
* READ CUSTOMER
        MOVE "read" TO SQL-ACTION
        MOVE "KEY IS CUS-CODE" TO SQL-OPTION
        CALL "WR-CUS" USING
        BY CONTENT SQL-ACTION
        BY REFERENCE CUS
        BY CONTENT SQL-OPTION
        BY REFERENCE WR-STATUS
**INVALID KEY
        IF WR-STATUS-INVALID-KEY
        DISPLAY "NO SUCH CUSTOMER"
        MOVE 1 TO END-FILE
        END-IF.

  READ-DETAIL.
        DISPLAY "PRODUCT CODE (0 = END) : ".
        ACCEPT PROD-CODE.
        IF PROD-CODE = 0
          MOVE 0
            TO REF-DET-STK(IND-DET)
          MOVE 0 TO END-FILE
        ELSE
          PERFORM READ-PROD-CODE.

  READ-PROD-CODE.
        MOVE 1 TO EXIST-PROD.
        MOVE PROD-CODE TO STK-CODE.
* READ STOCK
        MOVE "read" TO SQL-ACTION
        MOVE "KEY IS STK-CODE" TO SQL-OPTION
        CALL "WR-STK" USING
        BY CONTENT SQL-ACTION
        BY REFERENCE STK
        BY CONTENT SQL-OPTION
        BY REFERENCE WR-STATUS
**INVALID KEY
        IF WR-STATUS-INVALID-KEY
        MOVE 0 TO EXIST-PROD
        END-IF.
        IF EXIST-PROD = 0
          DISPLAY "NO SUCH PRODUCT"
        ELSE
           PERFORM UPDATE-ORD-DETAIL.

  UPDATE-ORD-DETAIL.
        MOVE 1 TO NEXT-DET.
        DISPLAY "QUANTITY ORDERED : "
          WITH NO ADVANCING
        ACCEPT ORD-QTY(IND-DET).
        PERFORM UNTIL
          (NEXT-DET < IND-DET
          AND REF-DET-STK(NEXT-DET) = PROD-CODE)
          OR IND-DET = NEXT-DET
          ADD 1 TO NEXT-DET
        END-PERFORM.
        IF IND-DET = NEXT-DET
          MOVE PROD-CODE
          TO REF-DET-STK(IND-DET)
          PERFORM UPDATE-CUS-HIST
          SET IND-DET UP BY 1
        ELSE
          DISPLAY "ERROR : ALREADY ORDERED".

  UPDATE-CUS-HIST.
        SET IND TO 1.
        PERFORM UNTIL
          REF-PURCH-STK(IND) = PROD-CODE
          OR REF-PURCH-STK(IND) = 0
          OR IND = 101
            SET IND UP BY 1
        END-PERFORM.
```

```
      IF IND = 101
      DISPLAY "ERR : HISTORY OVERFLOW"
        EXIT.
      IF REF-PURCH-STK(IND)
          = PROD-CODE
        ADD ORD-QTY(IND-DET) TO TOT(IND)
      ELSE
        MOVE PROD-CODE
        TO REF-PURCH-STK(IND)
        MOVE ORD-QTY(IND-DET) TO TOT(IND).

  LIST-ORD.
      DISPLAY "LIST OF ORDERS (BY NUMBER)".
      MOVE 0 TO ORD-CODE.
* START ORDERS KEY IS > ORD-CODE
      MOVE "start" TO SQL-ACTION
      MOVE "KEY IS > ORD-CODE" TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
**INVALID KEY
      IF WR-STATUS-INVALID-KEY
      MOVE 0 TO END-FILE
*  *NOT INVALID KEY
      ELSE
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM READ-ORD UNTIL END-FILE = 0.

  LIST-ORD-DATE.
      DISPLAY "LIST OF ORDERS (BY DATE)".
      DISPLAY "STARTING DATE".
      DISPLAY " DAY "
          WITH NO ADVANCING.
      ACCEPT ORDER-DD.
      DISPLAY " MONTH "
          WITH NO ADVANCING.
      ACCEPT ORDER-MM.
      DISPLAY " YEAR (YYYY) "
          WITH NO ADVANCING.
      ACCEPT ORDER-YYYY.
      MOVE ORDER-DATE TO ORD-DATE.
* START ORDERS KEY IS > ORD-DATE
      MOVE "start" TO SQL-ACTION
      MOVE "KEY IS > ORD-DATE" TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
**INVALID KEY
      IF WR-STATUS-INVALID-KEY
      MOVE 0 TO END-FILE
**NOT INVALID KEY
      ELSE
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM READ-ORD UNTIL END-FILE = 0.
  READ-ORD.
* READ ORDERS NEXT
      MOVE "read" TO SQL-ACTION
      MOVE " " TO SQL-OPTION
      CALL "WR-ORD" USING
      BY CONTENT SQL-ACTION
      BY REFERENCE ORD
      BY CONTENT SQL-OPTION
      BY REFERENCE WR-STATUS
**AT END
      IF WR-STATUS-AT-END
      MOVE 0 TO END-FILE
**NOT AT END
      ELSE
      DISPLAY "Order : " ORD-CODE " - " ORD-DATE
      DISPLAY "ORD-CUSTOMER " ORD-CUSTOMER
      MOVE ORD-DETAIL TO LIST-DETAIL
      SET IND-DET TO 1
      MOVE 1 TO END-DETAIL
      PERFORM DISPLAY-DETAIL
      UNTIL END-DETAIL = 0
      END-IF.

  INIT-HIST.
      SET IND TO 1.
      PERFORM UNTIL IND = 100
        MOVE 0 TO REF-PURCH-STK(IND)
        MOVE 0 TO TOT(IND)
        SET IND UP BY 1
      END-PERFORM.
      MOVE LIST-PURCHASE TO CUS-HIST.

  DISPLAY-DETAIL.
      IF IND-DET = 21
        MOVE 0 TO END-DETAIL
      ELSE
        IF REF-DET-STK(IND-DET) = 0
          MOVE 0 TO END-DETAIL
        ELSE
          MOVE REF-DET-STK(IND-DET) TO STK-CODE
* READ STOCK
          MOVE "read" TO SQL-ACTION
          MOVE "KEY IS STK-CODE" TO SQL-OPTION
          CALL "WR-STK" USING
          BY CONTENT SQL-ACTION
          BY REFERENCE STK
          BY CONTENT SQL-OPTION
          BY REFERENCE WR-STATUS
**INVALID KEY
          IF WR-STATUS-INVALID-KEY
          THEN
          DISPLAY "ERROR : UNKOWN PRODUCT"
**NOT INVALID KEY
          ELSE
          DISPLAY STK-NAME "  " ORD-QTY(IND-DET)
          END-IF
          SET IND-DET UP BY 1.
```

Appendix D

Statement Rewriting Conversion of the Case Study

This appendix contains the COBOL program obtained by transforming the case study program `order.cob` with respect to the Statement Rewriting conversion strategy. The source code fragments that have been transformed begin by a COBOL comment indicating the rewritten statement.

Here is the naming convention we used for the P2 paragraphs generation:

1. **Paragraphs names** The name of a COBOL paragraph cannot exceed 30 characters. So we used the following rules in order to minimize the names of the P2 generated paragraphs.
    - Cursors declaration: `P2-CURSORS-DECLARATION`
    - `OPEN` paragraph: `P2-OPEN-`*file*
    - `OPEN OUTPUT` paragraph: `P2-OPEN-OUT-`*file*
    - `CLOSE` paragraph: no paragraph
    - `START KEY IS =` paragraph: `P2-S-`*file*`-E-`*key*
    - `START KEY IS >` paragraph: `P2-S-`*file*`-G-`*key*
    - `START KEY IS =>` paragraph: `P2-S-`*file*`-N-`*key*
    - `READ NEXT` paragraph: `P2-R-`*file*`-NEXT`
    - `READ KEY IS` paragraph: `P2-R-`*file*`-K-`*key*
    - `WRITE` paragraph: `P2-WRITE-`*record*
    - `REWRITE` paragraph: `P2-REWRITE-`*record*
    - `DELETE` paragraph: `P2-DELETE-`*file*
2. **Cursor names** The name of a SQL cursor cannot exceed 18 characters. So we used the following rules in order to minimize the names of the cursors. The name of a cursor is of the form: (`O`|`E`|`G`|`N`)*key-name*.

- `O` stands for "order by"
- `E` stands for "equal to"
- `G` stands for "greater than"
- `N` stands for "not less than"
- *key-name* is an argument of the P2 conversion giving a name to each access key. (*key-name* contains less then 17 characters, and is compatible for SQL).

For instance, the *key-name* given for the access key `ORD-DATE` is `ORD_DATE`. So the cursor name corresponding to "greater than `ORD-DATE`" is called `GORD_DATE`.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. C-ORD.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.
EXEC SQL INCLUDE SQLCA END-EXEC.
EXEC SQL BEGIN DECLARE SECTION END-EXEC.
01 STK.
  02 STK-CODE PIC 9(5).
  02 STK-NAME PIC X(100).
  02 STK-LEVEL PIC 9(5).

01 ORD.
  02 ORD-CODE PIC 9(10).
  02 ORD-DATE PIC X(8).
  02 ORD-CUSTOMER PIC X(12).
  02 ORD-DETAIL PIC X(200).

01 CUS.
  02 CUS-CODE PIC X(12).
  02 CUS-DESCR PIC X(110).
  02 CUS-HIST PIC X(1000).

01 P2-STATUS PIC S9(9).
   88 P2-STATUS-NO-ERR VALUE 0.
   88 P2-STATUS-INVALID-KEY VALUE 1.
   88 P2-STATUS-AT-END VALUE 100.

01 P2-COUNTER PIC X(100).

01 LAST-CURSOR-CUSTOMER PIC X(100).

01 LAST-CURSOR-ORDERS PIC X(100).

01 LAST-CURSOR-STOCK PIC X(100).
EXEC SQL END DECLARE SECTION END-EXEC.

01 DESCRIPTION.
  02 NAME PIC X(20).
  02 ADDR PIC X(40).
  02 COMPANY PIC X(30).
  02 FUNCT PIC X(10).
  02 REC-DATE PIC X(10).

01 LIST-PURCHASE.
  02 PURCH OCCURS 100 TIMES
      INDEXED BY IND.
    03 REF-PURCH-STK PIC 9(5).
    03 TOT PIC 9(5).

01 LIST-DETAIL.
  02 DETAILS OCCURS 20 TIMES
      INDEXED BY IND-DET.
    03 REF-DET-STK PIC 9(5).
    03 ORD-QTY PIC 9(5).
```

```
  01 ORDER-DATE.
    02 ORDER-YYYY PIC X(4).
    02 ORDER-MM PIC X(2).
    02 ORDER-DD PIC X(2).
  01 CHOICE PIC X.
  01 END-FILE PIC 9.
  01 END-DETAIL PIC 9.
  01 EXIST-PROD PIC 9.
  01 PROD-CODE PIC 9(5).

  01 TOT-COMP PIC 9(5) COMP.
  01 QTY PIC 9(5) COMP.
  01 NEXT-DET PIC 99.

  PROCEDURE DIVISION.
  MAIN.
      PERFORM INIT.
      PERFORM PROCESS UNTIL CHOICE = 0.
      PERFORM CLOSING.
      STOP RUN.

  INIT.
*      *OPEN I-O CUSTOMER
      PERFORM P2-OPEN-CUSTOMER.
*      *OPEN I-O ORDERS
      PERFORM P2-OPEN-ORDERS.
*      *OPEN I-O STOCK
      PERFORM P2-OPEN-STOCK.

  PROCESS.
      DISPLAY "1 NEW CUSTOMER".
      DISPLAY "2 NEW STOCK".
      DISPLAY "3 NEW ORDER".
      DISPLAY "4 LIST OF CUSTOMERS".
      DISPLAY "5 LIST OF STOCKS".
      DISPLAY "6 LIST OF ORDERS (BY NUMBER)".
      DISPLAY "7 LIST OF ORDERS (BY DATE)".
      DISPLAY "8 DELETE CUSTOMER".
      DISPLAY "0 END".
      ACCEPT CHOICE.
      IF CHOICE = 1
        PERFORM NEW-CUS.
      IF CHOICE = 2
        PERFORM NEW-STK.
      IF CHOICE = 3
        PERFORM NEW-ORD.
      IF CHOICE = 4
        PERFORM LIST-CUS.
      IF CHOICE = 5
        PERFORM LIST-STK.
      IF CHOICE = 6
        PERFORM LIST-ORD.
      IF CHOICE = 7
        PERFORM LIST-ORD-DATE.
      IF CHOICE = 8
        PERFORM DELETE-CUS.
  CLOSING.
*      *CLOSE CUSTOMER
*      *nothing to do
        CONTINUE.
*      *CLOSE ORDERS
*      *nothing to do
```

```
        CONTINUE.
*      *CLOSE STOCK
*      *nothing to do
        CONTINUE.

  NEW-CUS.
      DISPLAY "NEW CUSTOMER :".
      DISPLAY "CUSTOMER CODE ?"
        WITH NO ADVANCING.
      ACCEPT CUS-CODE.

      DISPLAY "NAME DU CUSTOMER : "
        WITH NO ADVANCING.
      ACCEPT NAME.
      DISPLAY "ADDRESS OF CUSTOMER : "
        WITH NO ADVANCING.
      ACCEPT ADDR.
      DISPLAY "COMPANY OF CUSTOMER : "
        WITH NO ADVANCING.
      ACCEPT COMPANY.
      IF(COMPANY NOT= SPACE)
       DISPLAY "FUNCTION OF CUSTOMER : "
          WITH NO ADVANCING
        ACCEPT FUNCT
      ELSE
        MOVE SPACE TO FUNCT.
      DISPLAY "DATE : "
          WITH NO ADVANCING.
      ACCEPT REC-DATE.
      MOVE DESCRIPTION TO CUS-DESCR.
      PERFORM INIT-HIST.
*      *WRITE CUS
      PERFORM P2-WRITE-CUS
*      *INVALID KEY
      IF P2-STATUS-INVALID-KEY
           THEN
         DISPLAY "ERROR"
      END-IF.

  LIST-CUS.
      DISPLAY "LISTE DES CUSTOMERS".
*      *CLOSE CUSTOMER
*      *nothing to do
        CONTINUE.
*      *OPEN I-O CUSTOMER
      PERFORM P2-OPEN-CUSTOMER.
      MOVE 1 TO END-FILE.
      PERFORM READ-CUS
          UNTIL (END-FILE = 0).

  READ-CUS.
*      *READ CUSTOMER NEXT
      PERFORM P2-R-CUSTOMER-NEXT
*      *AT END
      IF P2-STATUS-AT-END
           THEN
         MOVE 0 TO END-FILE
*          *NOT AT END
      ELSE
      DISPLAY CUS-CODE
      DISPLAY CUS-DESCR
      DISPLAY CUS-HIST
```

```
          END-IF.

  DELETE-CUS.
      MOVE 1 TO END-FILE.
      PERFORM READ-CUS-CODE
          UNTIL END-FILE = 0.
      PERFORM DELETE-CUS-ORD.
*  *DELETE CUSTOMER
      PERFORM P2-DELETE-CUSTOMER.

  DELETE-CUS-ORD.
      MOVE CUS-CODE TO ORD-CUSTOMER.
      MOVE 0 TO END-FILE.
*  *READ ORDERS KEY IS ORD-CUSTOMER
      PERFORM P2-R-ORDERS-K-ORD-CUSTOMER
*  *INVALID KEY
      IF P2-STATUS-INVALID-KEY
        THEN
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM DELETE-ORDER
          UNTIL END-FILE = 1.
  DELETE-ORDER.
*  *DELETE ORDERS
      PERFORM P2-DELETE-ORDERS.
*  *READ ORDERS NEXT
      PERFORM P2-R-ORDERS-NEXT
*  *AT END
      IF P2-STATUS-AT-END
        THEN
      MOVE 1 TO END-FILE
*      *NOT AT END
      ELSE
      IF ORD-CUSTOMER NOT = CUS-CODE
      MOVE 1 TO END-FILE
      END-IF.

  NEW-STK.
      DISPLAY "NEW STOCK".
      DISPLAY "PRODUCT NUMBER : "
        WITH NO ADVANCING.
      ACCEPT STK-CODE.

      DISPLAY "NAME : "
          WITH NO ADVANCING.
      ACCEPT STK-NAME.

      DISPLAY "LEVEL : "
          WITH NO ADVANCING.
      ACCEPT STK-LEVEL.

*  *WRITE STK
      PERFORM P2-WRITE-STK
*  *INVALID KEY
      IF P2-STATUS-INVALID-KEY
        THEN
      DISPLAY "ERREUR "
      END-IF.

  LIST-STK.
      DISPLAY "LIST OF STOCKS ".
```

```
*  *CLOSE STOCK
*  *nothing to do
      CONTINUE.
*  *OPEN I-O STOCK
      PERFORM P2-OPEN-STOCK.

      MOVE 1 TO END-FILE.
      PERFORM READ-STK
          UNTIL END-FILE = 0.

  READ-STK.
*  *READ STOCK NEXT
      PERFORM P2-R-STOCK-NEXT
*  *AT END
      IF P2-STATUS-AT-END
        THEN
      MOVE 0 TO END-FILE
*       *NOT AT END
      ELSE
      DISPLAY STK-CODE
      DISPLAY STK-NAME
      DISPLAY STK-LEVEL
      END-IF.

  NEW-ORD.
      DISPLAY "NEW ORDER".
      DISPLAY "ORDER NUMBER : "
        WITH NO ADVANCING.
      ACCEPT ORD-CODE.

      DISPLAY "ORDER DATE".
      DISPLAY " DAY "
          WITH NO ADVANCING.
      ACCEPT ORDER-DD.
      DISPLAY " MONTH "
          WITH NO ADVANCING.
      ACCEPT ORDER-MM.
      DISPLAY " YEAR (YYYY) "
          WITH NO ADVANCING.
      ACCEPT ORDER-YYYY.
      MOVE ORDER-DATE TO ORD-DATE.
      MOVE 1 TO END-FILE.
      PERFORM READ-CUS-CODE
          UNTIL END-FILE = 0.
      MOVE CUS-DESCR TO DESCRIPTION.
      DISPLAY NAME.
      MOVE CUS-CODE TO ORD-CUSTOMER.
      MOVE CUS-HIST TO LIST-PURCHASE.

      SET IND-DET TO 1.
      MOVE 1 TO END-FILE.
      PERFORM READ-DETAIL
        UNTIL END-FILE = 0 OR IND-DET = 21.
      MOVE LIST-DETAIL TO ORD-DETAIL.

**WRITE ORD
      PERFORM P2-WRITE-ORD
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      DISPLAY "ERROR"
      END-IF.
```

```
      MOVE LIST-PURCHASE
        TO CUS-HIST.
**REWRITE CUS
      PERFORM P2-REWRITE-CUS
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      DISPLAY "ERROR CUS"
      END-IF.

  READ-CUS-CODE.
      DISPLAY "CUSTOMER NUMBER : "
        WITH NO ADVANCING.
      ACCEPT CUS-CODE.
      MOVE 0 TO END-FILE.
**READ CUSTOMER
      PERFORM P2-R-CUSTOMER-K-CUS-CODE
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      DISPLAY "NO SUCH CUSTOMER"
      MOVE 1 TO END-FILE
      END-IF.

  READ-DETAIL.
      DISPLAY "PRODUCT CODE (0 = END) : ".
      ACCEPT PROD-CODE.
      IF PROD-CODE = 0
        MOVE 0
          TO REF-DET-STK(IND-DET)
        MOVE 0 TO END-FILE
      ELSE
        PERFORM READ-PROD-CODE.

  READ-PROD-CODE.
      MOVE 1 TO EXIST-PROD.
      MOVE PROD-CODE TO STK-CODE.
**READ STOCK
      PERFORM P2-R-STOCK-K-STK-CODE
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      MOVE 0 TO EXIST-PROD
      END-IF.
      IF EXIST-PROD = 0
        DISPLAY "NO SUCH PRODUCT"
      ELSE
         PERFORM UPDATE-ORD-DETAIL.

  UPDATE-ORD-DETAIL.
      MOVE 1 TO NEXT-DET.
      DISPLAY "QUANTITY ORDERED : "
        WITH NO ADVANCING
      ACCEPT ORD-QTY(IND-DET).
      PERFORM UNTIL
        (NEXT-DET < IND-DET
        AND REF-DET-STK(NEXT-DET) = PROD-CODE)
        OR IND-DET = NEXT-DET
        ADD 1 TO NEXT-DET
      END-PERFORM.
      IF IND-DET = NEXT-DET
```

```
        MOVE PROD-CODE
        TO REF-DET-STK(IND-DET)
        PERFORM UPDATE-CUS-HIST
        SET IND-DET UP BY 1
      ELSE
        DISPLAY "ERROR : ALREADY ORDERED".

  UPDATE-CUS-HIST.
      SET IND TO 1.
      PERFORM UNTIL
        REF-PURCH-STK(IND) = PROD-CODE
        OR REF-PURCH-STK(IND) = 0
        OR IND = 101
          SET IND UP BY 1
      END-PERFORM.
      IF IND = 101
      DISPLAY "ERR : HISTORY OVERFLOW"
        EXIT.
      IF REF-PURCH-STK(IND)
          = PROD-CODE
        ADD ORD-QTY(IND-DET) TO TOT(IND)
      ELSE
        MOVE PROD-CODE
        TO REF-PURCH-STK(IND)
        MOVE ORD-QTY(IND-DET) TO TOT(IND).

  LIST-ORD.
      DISPLAY "LIST OF ORDERS (BY NUMBER)".
      MOVE 0 TO ORD-CODE.
**START ORDERS KEY IS > ORD-CODE
      PERFORM P2-S-ORDERS-G-ORD-CODE
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      MOVE 0 TO END-FILE
*  *NOT INVALID KEY
      ELSE
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM READ-ORD UNTIL END-FILE = 0.

  LIST-ORD-DATE.
      DISPLAY "LIST OF ORDERS (BY DATE)".
      DISPLAY "STARTING DATE".
      DISPLAY " DAY "
          WITH NO ADVANCING.
      ACCEPT ORDER-DD.
      DISPLAY " MONTH "
          WITH NO ADVANCING.
      ACCEPT ORDER-MM.
      DISPLAY " YEAR (YYYY) "
          WITH NO ADVANCING.
      ACCEPT ORDER-YYYY.
      MOVE ORDER-DATE TO ORD-DATE.
**START ORDERS KEY IS > ORD-DATE
      PERFORM P2-S-ORDERS-G-ORD-DATE
**INVALID KEY
      IF P2-STATUS-INVALID-KEY
      THEN
      MOVE 0 TO END-FILE
**NOT INVALID KEY
      ELSE
```

```
      MOVE 1 TO END-FILE
      END-IF.
      PERFORM READ-ORD UNTIL END-FILE = 0.
  READ-ORD.
**READ ORDERS NEXT
      PERFORM P2-R-ORDERS-NEXT
**AT END
      IF P2-STATUS-AT-END
      THEN
      MOVE 0 TO END-FILE
**NOT AT END
      ELSE
      DISPLAY "Order : " ORD-CODE " - " ORD-DATE
      DISPLAY "ORD-CUSTOMER " ORD-CUSTOMER
      MOVE ORD-DETAIL TO LIST-DETAIL
      SET IND-DET TO 1
      MOVE 1 TO END-DETAIL
      PERFORM DISPLAY-DETAIL
      UNTIL END-DETAIL = 0
      END-IF.

  INIT-HIST.
      SET IND TO 1.
      PERFORM UNTIL IND = 100
        MOVE 0 TO REF-PURCH-STK(IND)
        MOVE 0 TO TOT(IND)
        SET IND UP BY 1
      END-PERFORM.
      MOVE LIST-PURCHASE TO CUS-HIST.

  DISPLAY-DETAIL.
      IF IND-DET = 21
        MOVE 0 TO END-DETAIL
      ELSE
        IF REF-DET-STK(IND-DET) = 0
          MOVE 0 TO END-DETAIL
        ELSE
          MOVE REF-DET-STK(IND-DET) TO STK-CODE
**READ STOCK
          PERFORM P2-R-STOCK-K-STK-CODE
**INVALID KEY
          IF P2-STATUS-INVALID-KEY
          THEN
          DISPLAY "ERROR : UNKOWN PRODUCT"
**NOT INVALID KEY
          ELSE
          DISPLAY STK-NAME "  " ORD-QTY(IND-DET)
          END-IF
          SET IND-DET UP BY 1.

  P2-STRATEGY SECTION.
   P2-CURSORS-DECLARATION.
   EXEC SQL
      DECLARE OCUS_CODE CURSOR FOR
      SELECT CUS_CODE,
             CUS_DESCR,
             CUS_HIST
      FROM CUSTOMER
      ORDER BY CUS_CODE
     END-EXEC.

    EXEC SQL
   DECLARE ECUS_CODE CURSOR FOR
    SELECT CUS_CODE,
           CUS_DESCR,
           CUS_HIST
    FROM CUSTOMER
    WHERE CUS_CODE = :CUS.CUS-CODE
    ORDER BY CUS_CODE
 END-EXEC.

  EXEC SQL
   DECLARE GCUS_CODE CURSOR FOR
    SELECT CUS_CODE,
           CUS_DESCR,
           CUS_HIST
    FROM CUSTOMER
    WHERE CUS_CODE > :CUS.CUS-CODE
    ORDER BY CUS_CODE
 END-EXEC.

  EXEC SQL
    DECLARE NCUS_CODE CURSOR FOR
    SELECT CUS_CODE,
           CUS_DESCR,
           CUS_HIST
    FROM CUSTOMER
    WHERE CUS_CODE >= :CUS.CUS-CODE
    ORDER BY CUS_CODE
   END-EXEC.


  EXEC SQL
    DECLARE OORD_CODE CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    ORDER BY ORD_CODE
   END-EXEC.

  EXEC SQL
   DECLARE EORD_CODE CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CODE = :ORD.ORD-CODE
    ORDER BY ORD_CODE
 END-EXEC.

  EXEC SQL
   DECLARE GORD_CODE CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CODE > :ORD.ORD-CODE
    ORDER BY ORD_CODE
 END-EXEC.
```

```
  EXEC SQL
    DECLARE NORD_CODE CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CODE >= :ORD.ORD-CODE
    ORDER BY ORD_CODE
   END-EXEC.

 EXEC SQL
    DECLARE OORD_CUST CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    ORDER BY ORD_CUSTOMER
   END-EXEC.

  EXEC SQL
   DECLARE EORD_CUST CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CUSTOMER = :ORD.ORD-CUSTOMER
    ORDER BY ORD_CUSTOMER
 END-EXEC.

  EXEC SQL
   DECLARE GORD_CUST CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CUSTOMER > :ORD.ORD-CUSTOMER
    ORDER BY ORD_CUSTOMER
 END-EXEC.

  EXEC SQL
    DECLARE NORD_CUST CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    WHERE ORD_CUSTOMER >= :ORD.ORD-CUSTOMER
    ORDER BY ORD_CUSTOMER
   END-EXEC.

 EXEC SQL
    DECLARE OORD_DATE CURSOR FOR
    SELECT ORD_CODE,
           ORD_DATE,
           ORD_CUSTOMER,
           ORD_DETAIL
    FROM ORDERS
    ORDER BY ORD_DATE
  END-EXEC.

 EXEC SQL
  DECLARE EORD_DATE CURSOR FOR
   SELECT ORD_CODE,
          ORD_DATE,
          ORD_CUSTOMER,
          ORD_DETAIL
   FROM ORDERS
   WHERE ORD_DATE = :ORD.ORD-DATE
   ORDER BY ORD_DATE
END-EXEC.

 EXEC SQL
  DECLARE GORD_DATE CURSOR FOR
   SELECT ORD_CODE,
          ORD_DATE,
          ORD_CUSTOMER,
          ORD_DETAIL
   FROM ORDERS
   WHERE ORD_DATE > :ORD.ORD-DATE
   ORDER BY ORD_DATE
END-EXEC.

 EXEC SQL
   DECLARE NORD_DATE CURSOR FOR
   SELECT ORD_CODE,
          ORD_DATE,
          ORD_CUSTOMER,
          ORD_DETAIL
   FROM ORDERS
   WHERE ORD_DATE >= :ORD.ORD-DATE
   ORDER BY ORD_DATE
  END-EXEC.


 EXEC SQL
   DECLARE OSTK_CODE CURSOR FOR
   SELECT STK_CODE,
          STK_NAME,
          STK_LEVEL
   FROM STOCK
   ORDER BY STK_CODE
  END-EXEC.

 EXEC SQL
  DECLARE ESTK_CODE CURSOR FOR
   SELECT STK_CODE,
          STK_NAME,
          STK_LEVEL
   FROM STOCK
   WHERE STK_CODE = :STK.STK-CODE
   ORDER BY STK_CODE
END-EXEC.

 EXEC SQL
  DECLARE GSTK_CODE CURSOR FOR
   SELECT STK_CODE,
          STK_NAME,
          STK_LEVEL
   FROM STOCK
   WHERE STK_CODE > :STK.STK-CODE
```

```
   ORDER BY STK_CODE
END-EXEC.

 EXEC SQL
   DECLARE NSTK_CODE CURSOR FOR
   SELECT STK_CODE,
          STK_NAME,
          STK_LEVEL
   FROM STOCK
   WHERE STK_CODE >= :STK.STK-CODE
   ORDER BY STK_CODE
  END-EXEC.

P2-OPEN-STOCK.
PERFORM CLOSE-LAST-CURSOR-STOCK.
    MOVE "OSTK_CODE" TO LAST-CURSOR-STOCK.
    EXEC SQL
        OPEN OSTK_CODE
    END-EXEC.

P2-OPEN-ORDERS.
PERFORM CLOSE-LAST-CURSOR-ORDERS.
    MOVE "OORD_CODE" TO LAST-CURSOR-ORDERS.
    EXEC SQL
        OPEN OORD_CODE
    END-EXEC.

P2-OPEN-CUSTOMER.
PERFORM CLOSE-LAST-CURSOR-CUSTOMER.
    MOVE "OCUS_CODE" TO LAST-CURSOR-CUSTOMER.
    EXEC SQL
        OPEN OCUS_CODE
    END-EXEC.

P2-S-ORDERS-G-ORD-CODE.
PERFORM CLOSE-LAST-CURSOR-ORDERS
    EXEC SQL
       SELECT COUNT(*)
       INTO :P2-COUNTER
       FROM ORDERS
       WHERE ORD_CODE > :ORD.ORD-CODE
    END-EXEC
    IF (SQLCODE NOT = 0)
     THEN
      MOVE SQLCODE TO P2-STATUS
     ELSE
      IF (P2-COUNTER = 0)
      THEN
         SET P2-STATUS-INVALID-KEY TO TRUE
      ELSE
         EXEC SQL
           OPEN GORD_CODE
         END-EXEC
         MOVE "GORD_CODE"
              TO LAST-CURSOR-ORDERS
         MOVE SQLCODE TO P2-STATUS
      END-IF
    END-IF.

P2-S-ORDERS-G-ORD-DATE.
PERFORM CLOSE-LAST-CURSOR-ORDERS
    EXEC SQL
```

```
      SELECT COUNT(*)
      INTO :P2-COUNTER
      FROM ORDERS
      WHERE ORD_DATE > :ORD.ORD-DATE
   END-EXEC
   IF (SQLCODE NOT = 0)
    THEN
     MOVE SQLCODE TO P2-STATUS
    ELSE
     IF (P2-COUNTER = 0)
     THEN
        SET P2-STATUS-INVALID-KEY TO TRUE
     ELSE
        EXEC SQL
          OPEN GORD_DATE
        END-EXEC
        MOVE "GORD_DATE"
             TO LAST-CURSOR-ORDERS
        MOVE SQLCODE TO P2-STATUS
     END-IF
   END-IF.

P2-R-STOCK-NEXT.
IF(LAST-CURSOR-STOCK = "OSTK_CODE")
     THEN
           EXEC SQL
              FETCH OSTK_CODE
              INTO  :STK.STK-CODE  ,
                    :STK.STK-NAME  ,
                    :STK.STK-LEVEL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-STOCK = "ESTK_CODE")
     THEN
           EXEC SQL
              FETCH ESTK_CODE
              INTO  :STK.STK-CODE  ,
                    :STK.STK-NAME  ,
                    :STK.STK-LEVEL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-STOCK = "GSTK_CODE")
     THEN
           EXEC SQL
              FETCH GSTK_CODE
              INTO  :STK.STK-CODE  ,
                    :STK.STK-NAME  ,
                    :STK.STK-LEVEL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-STOCK = "NSTK_CODE")
     THEN
           EXEC SQL
              FETCH NSTK_CODE
              INTO  :STK.STK-CODE  ,
                    :STK.STK-NAME  ,
                    :STK.STK-LEVEL
            END-EXEC
    END-IF
    MOVE SQLCODE TO P2-STATUS.

P2-R-ORDERS-NEXT.
IF(LAST-CURSOR-ORDERS = "OORD_CODE")
     THEN
           EXEC SQL
              FETCH OORD_CODE
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_CODE")
     THEN
           EXEC SQL
              FETCH EORD_CODE
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_CODE")
     THEN
           EXEC SQL
              FETCH GORD_CODE
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_CODE")
     THEN
           EXEC SQL
              FETCH NORD_CODE
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
   IF(LAST-CURSOR-ORDERS = "OORD_CUST")
     THEN
           EXEC SQL
              FETCH OORD_CUST
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_CUST")
     THEN
           EXEC SQL
              FETCH EORD_CUST
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_CUST")
     THEN
```

```
          EXEC SQL
             FETCH GORD_CUST
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_CUST")
     THEN
          EXEC SQL
             FETCH NORD_CUST
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
   IF(LAST-CURSOR-ORDERS = "OORD_DATE")
     THEN
          EXEC SQL
             FETCH OORD_DATE
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_DATE")
     THEN
          EXEC SQL
             FETCH EORD_DATE
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_DATE")
     THEN
          EXEC SQL
             FETCH GORD_DATE
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_DATE")
     THEN
          EXEC SQL
             FETCH NORD_DATE
             INTO  :ORD.ORD-CODE  ,
                   :ORD.ORD-DATE  ,
                   :ORD.ORD-CUSTOMER  ,
                   :ORD.ORD-DETAIL
           END-EXEC
    END-IF
    MOVE SQLCODE TO P2-STATUS.

P2-R-CUSTOMER-NEXT.
IF(LAST-CURSOR-CUSTOMER = "OCUS_CODE")
     THEN
          EXEC SQL
             FETCH OCUS_CODE
             INTO  :CUS.CUS-CODE  ,
                   :CUS.CUS-DESCR  ,
                   :CUS.CUS-HIST
           END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "ECUS_CODE")
     THEN
          EXEC SQL
             FETCH ECUS_CODE
             INTO  :CUS.CUS-CODE  ,
                   :CUS.CUS-DESCR  ,
                   :CUS.CUS-HIST
           END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "GCUS_CODE")
     THEN
          EXEC SQL
             FETCH GCUS_CODE
             INTO  :CUS.CUS-CODE  ,
                   :CUS.CUS-DESCR  ,
                   :CUS.CUS-HIST
           END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "NCUS_CODE")
     THEN
          EXEC SQL
             FETCH NCUS_CODE
             INTO  :CUS.CUS-CODE  ,
                   :CUS.CUS-DESCR  ,
                   :CUS.CUS-HIST
           END-EXEC
    END-IF
    MOVE SQLCODE TO P2-STATUS.

P2-R-STOCK-K-STK-CODE.
PERFORM CLOSE-LAST-CURSOR-STOCK.
    EXEC SQL
      SELECT COUNT(*)
      INTO :P2-COUNTER
      FROM STOCK
      WHERE STK_CODE = :STK.STK-CODE
    END-EXEC.
    IF (SQLCODE NOT = 0)
    THEN
      MOVE SQLCODE TO P2-STATUS
      ELSE
        IF (P2-COUNTER = 0)
         THEN
            SET P2-STATUS-INVALID-KEY TO TRUE
         ELSE
            EXEC SQL
              OPEN NSTK_CODE
            END-EXEC
            MOVE "NSTK_CODE"
                TO LAST-CURSOR-STOCK
            EXEC SQL
              FETCH NSTK_CODE
              INTO  :STK.STK-CODE  ,
                    :STK.STK-NAME  ,
                :STK.STK-LEVEL
        END-EXEC
        MOVE SQLCODE TO P2-STATUS
      END-IF
    END-IF.

P2-R-ORDERS-K-ORD-CUSTOMER.
PERFORM CLOSE-LAST-CURSOR-ORDERS.
    EXEC SQL
      SELECT COUNT(*)
      INTO :P2-COUNTER
      FROM ORDERS
      WHERE ORD_CUSTOMER = :ORD.ORD-CUSTOMER
    END-EXEC.
    IF (SQLCODE NOT = 0)
    THEN
      MOVE SQLCODE TO P2-STATUS
      ELSE
        IF (P2-COUNTER = 0)
         THEN
            SET P2-STATUS-INVALID-KEY TO TRUE
         ELSE
            EXEC SQL
              OPEN NORD_CUST
            END-EXEC
            MOVE "NORD_CUST"
                TO LAST-CURSOR-ORDERS
            EXEC SQL
              FETCH NORD_CUST
              INTO  :ORD.ORD-CODE  ,
                    :ORD.ORD-DATE  ,
                    :ORD.ORD-CUSTOMER  ,
                    :ORD.ORD-DETAIL
            END-EXEC
            MOVE SQLCODE TO P2-STATUS
        END-IF
    END-IF.

P2-R-CUSTOMER-K-CUS-CODE.
PERFORM CLOSE-LAST-CURSOR-CUSTOMER.
    EXEC SQL
      SELECT COUNT(*)
      INTO :P2-COUNTER
      FROM CUSTOMER
      WHERE CUS_CODE = :CUS.CUS-CODE
    END-EXEC.
    IF (SQLCODE NOT = 0)
    THEN
      MOVE SQLCODE TO P2-STATUS
      ELSE
        IF (P2-COUNTER = 0)
         THEN
            SET P2-STATUS-INVALID-KEY TO TRUE
         ELSE
            EXEC SQL
              OPEN NCUS_CODE
            END-EXEC
            MOVE "NCUS_CODE"
                TO LAST-CURSOR-CUSTOMER
            EXEC SQL
              FETCH NCUS_CODE
              INTO  :CUS.CUS-CODE  ,
                    :CUS.CUS-DESCR  ,
                    :CUS.CUS-HIST
            END-EXEC
            MOVE SQLCODE TO P2-STATUS
        END-IF
    END-IF.

P2-WRITE-CUS.
EXEC SQL
        INSERT INTO CUSTOMER
        VALUES ( :CUS.CUS-CODE  ,
                 :CUS.CUS-DESCR  ,
                 :CUS.CUS-HIST )
      END-EXEC
      MOVE SQLCODE TO P2-STATUS.

P2-WRITE-ORD.
EXEC SQL
        INSERT INTO ORDERS
        VALUES ( :ORD.ORD-CODE  ,
                 :ORD.ORD-DATE  ,
                 :ORD.ORD-CUSTOMER  ,
                 :ORD.ORD-DETAIL )
      END-EXEC
      MOVE SQLCODE TO P2-STATUS.

P2-WRITE-STK.
EXEC SQL
        INSERT INTO STOCK
        VALUES ( :STK.STK-CODE  ,
                 :STK.STK-NAME  ,
                 :STK.STK-LEVEL )
      END-EXEC
      MOVE SQLCODE TO P2-STATUS.

P2-REWRITE-CUS.
EXEC SQL
       UPDATE CUSTOMER
       SET CUS_CODE = :CUS.CUS-CODE ,
           CUS_DESCR = :CUS.CUS-DESCR ,
           CUS_HIST = :CUS.CUS-HIST
       WHERE CUS_CODE = :CUS.CUS-CODE
     END-EXEC.
     MOVE SQLCODE TO P2-STATUS.

P2-DELETE-ORDERS.
EXEC SQL
        DELETE
        FROM ORDERS
        WHERE ORD_CODE = :ORD.ORD-CODE
     END-EXEC.
     MOVE SQLCODE TO P2-STATUS.

P2-DELETE-CUSTOMER.
EXEC SQL
        DELETE
        FROM CUSTOMER
        WHERE CUS_CODE = :CUS.CUS-CODE
     END-EXEC.
     MOVE SQLCODE TO P2-STATUS.

CLOSE-LAST-CURSOR-CUSTOMER.
```

```
    IF(LAST-CURSOR-CUSTOMER = "OCUS_CODE")
        THEN
          EXEC SQL
             CLOSE OCUS_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "ECUS_CODE")
        THEN
          EXEC SQL
             CLOSE ECUS_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "GCUS_CODE")
        THEN
          EXEC SQL
             CLOSE GCUS_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-CUSTOMER = "NCUS_CODE")
        THEN
          EXEC SQL
             CLOSE NCUS_CODE
          END-EXEC
    END-IF.

CLOSE-LAST-CURSOR-ORDERS.
    IF(LAST-CURSOR-ORDERS = "OORD_CODE")
        THEN
          EXEC SQL
             CLOSE OORD_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_CODE")
        THEN
          EXEC SQL
             CLOSE EORD_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_CODE")
        THEN
          EXEC SQL
             CLOSE GORD_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_CODE")
        THEN
          EXEC SQL
             CLOSE NORD_CODE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "OORD_CUST")
        THEN
          EXEC SQL
             CLOSE OORD_CUST
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_CUST")
        THEN
          EXEC SQL
             CLOSE EORD_CUST
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_CUST")
        THEN
          EXEC SQL
             CLOSE GORD_CUST
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_CUST")
        THEN
          EXEC SQL
             CLOSE NORD_CUST
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "OORD_DATE")
        THEN
          EXEC SQL
             CLOSE OORD_DATE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "EORD_DATE")
        THEN
          EXEC SQL
             CLOSE EORD_DATE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "GORD_DATE")
        THEN
          EXEC SQL
             CLOSE GORD_DATE
          END-EXEC
    END-IF
    IF(LAST-CURSOR-ORDERS = "NORD_DATE")
        THEN
          EXEC SQL
             CLOSE NORD_DATE
          END-EXEC
    END-IF.

CLOSE-LAST-CURSOR-STOCK.
   IF(LAST-CURSOR-STOCK = "OSTK_CODE")
        THEN
          EXEC SQL
             CLOSE OSTK_CODE
          END-EXEC
      END-IF
   IF(LAST-CURSOR-STOCK = "ESTK_CODE")
        THEN
          EXEC SQL
             CLOSE ESTK_CODE
          END-EXEC
   END-IF
   IF(LAST-CURSOR-STOCK = "GSTK_CODE")
        THEN
          EXEC SQL
             CLOSE GSTK_CODE
          END-EXEC
   END-IF
   IF(LAST-CURSOR-STOCK = "NSTK_CODE")
        THEN
          EXEC SQL
             CLOSE NSTK_CODE
          END-EXEC
```