DOCTOR OF SCIENCES

Program analysis and transformation for data-intensive system evolution

Cleve, Anthony

Award date:
2009

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix
Faculté d'Informatique
Namur, Belgique

Program Analysis and Transformation for Data-Intensive System Evolution

Anthony Cleve

October 2009

Thèse présentée en vue de l'obtention du grade de Docteur en Sciences
(orientation Informatique)

© Presses universitaires de Namur, 2009
Rempart de la Vierge, 13
B - 5000 Namur (Belgique)

Imprimé en Belgique
ISBN : 978-2-87037-655-3
Dépôt légal: D/2009/1881/42

PhD Committee

Prof. Jean-Marie Jacquet, University of Namur (President)
Prof. Jean-Luc Hainaut, University of Namur (Promotor)
Prof. Ralf Lämmel, Universität Koblenz-Landau (External Reviewer)
Prof. Kim Mens, University of Louvain, Belgium (External Reviewer)
Prof. Vincent Englebert, University of Namur (Internal Reviewer)
Prof. Wim Vanhoof, University of Namur (Internal Reviewer)

The public PhD defence was held on the 29th October 2009, at the University of Namur.

Abstract

Data-intensive software systems are generally made of a database (sometimes in the form of a set of files) and a collection of application programs in strong interaction with the former. They constitute critical assets in most enterprises, since they support business activities in all production and management domains. Data-intensive systems form most of the so-called legacy systems: they typically are one or more decade old, they are very large, heterogeneous and highly complex. Many of them *significantly resist modifications and change* due to the lack of documentation, to the use of ageing technologies and to inflexible architectures. Therefore, the evolution of data-intensive systems clearly calls for automated support.

This thesis particularly explores the use of automated *program analysis* and *transformation* techniques in support to the evolution of the database component of the system. The program analysis techniques aim to ease the database evolution process, by helping the developers to understand the data structures that are to be changed, despite the lack of precise and up-to-date documentation. The objective of the program transformation techniques is to support the adaptation of the application programs to the new database. This adaptation process is studied in the context of two realistic database evolution scenarios, namely database platform migration and database schema refactoring.

Résumé

Les systèmes d'information sont généralement composés d'une base de données (parfois sous la forme d'un ensemble de fichiers) et d'une collection de programmes d'application en forte interaction avec celle-ci. Ces systèmes constituent des composants critiques dans la plupart des entreprises et organisations, car ils supportent leurs activités dans tous les domaines de production et de gestion. Les systèmes d'information forment souvent ce que l'on appelle des systèmes hérités: ils ont été développés il y a plus de dix ans, sont très volumineux, hétérogènes et hautement complexes. La plupart d'entre eux résistent fortement au changement, de part le manque de documentation, l'utilisation de technologies obsolètes et d'architectures peu flexibles. C'est pourquoi l'évolution des systèmes d'information nécessite un plus grand support automatisé.

Cette thèse se propose d'explorer l'utilisation de techniques d'analyse et de transformation automatique de programmes, comme support à l'évolution de la base de données d'un système d'information. Les techniques d'analyse ont pour but de faciliter le processus d'évolution de la base de données, en aidant les développeurs à comprendre les structures de données qui doivent évoluer, malgré le manque de documentation. L'objectif des techniques de transformation de programmes et de supporter l'adaptation des programmes d'applications à la nouvelle base de données. Ce processus d'adaptation est étudié dans le contexte de deux scénarios réalistes d'évolution: la migration de la base de données vers une nouvelle plateforme et la restructuration de son schéma.

Preface

It's a sign of mediocrity when you demonstrate gratitude with moderation.
– Roberto Benigni

The PhD student paradox is the following. On the one hand, doing a PhD thesis is typically an isolating experience, where the student often feels alone, lost and misunderstood. On the other hand, accomplishing such a work is impossible without the support and encouragements of numerous people, including colleagues, family members and friends. This the reason why I would like to take the time to thank all the persons without whom I would have never completed this thesis.

First of all, I would like to express my gratitude to my supervisor, Prof. Jean-Luc Hainaut, who gave me the opportunity to perform this PhD research. Jean-Luc, thank you for your trust in me. Thank you for teaching me to learn and to teach. Thank you for providing me with a lot of freedom in my work. Thank you for allowing me to travel a lot and, consequently, to meet so many people.

I wish to warmly thank the other members of my PhD committee, Prof. Jean-Marie Jacquet, Prof. Ralf Lämmel, Prof Kim Mens, Prof. Vincent Englebert and Prof. Wim Vanhoof for their very detailed comments on the preliminary version of this thesis. Their constructive feedback and judicious advices allowed me to significantly improve the quality of the dissertation. Obviously, any remaining errors and inconsistencies are mine.

Most of this research would have been impossible without the cooperation of ReVeR, our industrial partner. This partnership allowed me to improve and validate my research results in the context of real-life reverse engineering and migration projects. I would like to sincerely thank Jean Henrard, for playing the implicit role of co-supervisor and for helping me to convert naive prototypes into scalable tools. Special thanks are also due to Jean-Marc Hick for his coaching on the topic of schema mapping, to Didier Roland for his project leadership, to Vincent Ciselet for his great sense of humor, and to Dominique Orban for his enthusiasm and trust.

The SEN1 group of CWI in Amsterdam has been another essential research partner of this work. I am very grateful to Paul Klint, who welcomed me two times in his research team. Working in this group has been an excellent experience, from which my taste for research definitely originates. Many thanks to all the (former) SEN1 members, and in particular to Paul Klint, Arie van Deursen, Mark van den Brand, Ralf Lämmel, Jurgen Vinju, Magiel Bruntink, Tijs van der storm, Rob

Economopoulos and Diego Ordonez. Special thanks to Arie van Deursen for the enthusiastic supervision of my Master's thesis internship in 2003. I also thank Niels Veerman and Steven Klusener, from the Free University of Amsterdam, for their precious cooperation at that time.

I would like to thank all the members of the Computer Science Faculty of the University of Namur, for providing me with such a pleasant working atmosphere. Special thanks to Anne-France, Virginie, Jean-Roch, Ravi and Jonathan, my (former) colleagues at the Laboratory of Database Engineering, for their positive attitude at work and their encouragements. I thank all the *coffee-break people* for the funny discussions we had there. Many thanks to Vincent, Laurent and Patrick, who contributed to strengthening my motivation when needed. I am grateful to Isabelle, Gaby, Jean-Roch and Vincent, for their friendship. I thank all the members of the MoVES interuniversity network. Special thanks to Arnaud Hubaux, Ragnhild Van Der Straeten, Dirk Deridder, Kim Mens, Sergio Castro, Tom Mens, Anne Keller, and Olaf Muliawan, for their cooperation on the particular topic of inconsistency management.

I would like to thank all my close friends for their patience, their encouragements and their enthusiasm during our rare, yet intense distracting activities. Special thanks are due to Loup, Valet, Bertrand and all the Stiltwalkers of Namur.

Last but not least, none of this would have been possible without the invaluable support of my wonderful family. I thank my parents, Antoine, Loreta and Marc, who taught me the most essential things. I warmly thank Louis and Marie-Pierre for their precious assistance in so many occasions, especially during the last months of writing. I am extremely grateful to my wife, Julie, for her love, her patience and her encouragements all along the road. Julie, I sincerely apologize for my long and frequent absences, particularly when I was home. I hope you will be able to forgive me one day. I thank Tristan and Naomé for their beautiful smiles, that erase all my doubts in a second. I thank Sacha for keeping me awake day and night, which helped me to submit this dissertation in due time.

Anthony Cleve, October 2009.

This research was supported by the Belgian Région Wallonne and the European Social Fund, in the context of the RISTART project. Partial support was also received from the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy, in the context of the MoVES project.

RÉGION WALLONNE

European Union
European Social Fund

Belgian
Science Policy

à Julie, Tristan, Naomé et Sacha

Contents

List of Figures **v**

List of Tables **ix**

List of Acronyms **xi**

1 Introduction **1**

1.1 Data-intensive software systems 1

1.2 The database component of software systems 1

1.3 On the complexity of data-intensive software systems 2

1.4 Evolution, maintenance and reverse engineering 3

1.5 System reengineering and migration 3

1.6 Goals of the Thesis 4

1.7 Research questions 4

1.8 Outline of the Thesis 5

1.9 Publications 7

I Research Domain **9**

2 Conceptual Background **11**

2.1 Database design 11

2.2 Data(base) reverse engineering 12

2.3 The Generic Entity-Relationship model 13

2.4 The transformational approach 18

3 A Framework for Data-Intensive System Evolution **25**

3.1 The nature of consistency relationships 25

3.2 Classification of database evolution scenarios 33

3.3 Database evolution processes 38

3.4 Thesis scope, contributions and outline revisited 39

II The System Migration Problem 43

4 Strategies for Data-Intensive System Migration 45
4.1 System migration: State of the Art 45
4.2 Migration reference model 49
4.3 Schema conversion 52
4.4 Data conversion 56
4.5 Program conversion 58
4.6 Strategies comparison 64
4.7 Conclusions 68

III Program Analysis for Database Reverse Engineering 69

5 Static Dependency Analysis 71
5.1 Introduction 71
5.2 Basic concepts 72
5.3 Problem statement 75
5.4 Slicing with embedded DML 76
5.5 Slicing with call-based DML 79
5.6 Tool support 83
5.7 Industrial applications 85
5.8 Related work 88
5.9 Conclusions 89

6 Dynamic Analysis of SQL Queries 91
6.1 Introduction 91
6.2 Static VS dynamic SQL 95
6.3 Applications of SQL statement analysis 98
6.4 SQL statement capturing techniques 101
6.5 Evaluation and applicability of SQL capturing techniques 107
6.6 Aspect-based dynamic analysis 108
6.7 SQL trace processing 113
6.8 Application to database reverse engineering 116
6.9 Initial experiment 120
6.10 Conclusions and perspectives 129

IV Adapting Programs to Database Platform Migration 131

7 Migrating Standard Files to a Relational Database 133
7.1 COBOL file management 133
7.2 Wrapper-based program conversion 136
7.3 Statement rewriting program conversion 139
7.4 COBOL-to-SQL translation 140

7.5 About correctness 151
7.6 Tool support 154
7.7 Initial case studies 156
7.8 Conclusions 157

8 Migrating a CODASYL Database to a Relational Database 159
8.1 CODASYL data management 159
8.2 Migration methodology 162
8.3 Wrapper-based program conversion 164
8.4 CODASYL-to-SQL translation 172
8.5 Tool Support 210
8.6 Related Work 214
8.7 Conclusions 215

9 Industrial Migration Projects 217
9.1 Project 1: IDS/II to DB2 217
9.2 Project 2: IDS/II to DB2 with a refined methodology 220
9.3 Evaluation 225
9.4 Conclusions and lessons learned 226

V Adapting Programs to Database Schema Change 229

10 A Co-transformational Approach to Schema Refactoring 231
10.1 Introduction 231
10.2 General approach 232
10.3 The LDA language 233
10.4 Schema transformations 237
10.5 Program adaptation by co-transformations 237
10.6 Co-transformation rules 242
10.7 Applications 259
10.8 Tool support 266
10.9 Related work 266
10.10Discussion 269
10.11Conclusions 271

VI Conclusions 273

11 Conclusions 275
11.1 Summary of the contributions 275
11.2 Lessons learned 278
11.3 Open issues and future challenges 279

Bibliography 283

Appendices **297**

A COBOL File Handling Statements **297**

A.1 `OPEN` statement . . . 297
A.2 `CLOSE` statement . . . 297
A.3 `START` statement . . . 297
A.4 `READ` statement . . . 298
A.5 `WRITE` statement . . . 300
A.6 `REWRITE` statement . . . 300
A.7 `DELETE` statement . . . 301

B LDA Language: Syntax and Partial Semantics **303**

B.1 Concrete syntax . . . 303
B.2 Semantics . . . 305

List of Figures

1.1 General structure of the thesis. 6

2.1 Standard database design processes. 11
2.2 Standard database reverse engineering processes. 12
2.3 Sample GER conceptual schema. 15
2.4 Sample GER logical schema. 17
2.5 Sample GER physical schema. 19
2.6 Schema transformation defined as a couple of mappings. 20
2.7 Structural mapping of a schema transformation. 20

3.1 Consistency relationships in data-intensive applications. 26
3.2 Semantically equivalent conceptual, logical and physical schemas. . . 28
3.3 Illustration of semantics-decreasing logical design. 29
3.4 DDL code corresponding to physical schema of Figure 3.2. 30
3.5 Consistency of a SQL query w.r.t. the underlying DML syntax. . . . 31
3.6 Consistency of a SQL query w.r.t. the underlying logical schema. . . 32
3.7 Consistency of an `insert` query w.r.t. an implicit foreign key. 33
3.8 Database evolution scenarios classified according to three dimensions. 35
3.9 Examples of S^+, S^- and $S^=$ transformations. 36
3.10 Classification of the database evolution scenarios studied in this thesis. 40
3.11 Thesis chapters VS database evolution scenarios and processes . . . 41

4.1 Overall view of the *database-first* system migration process 50
4.2 The six reference IS migration strategies 52
4.3 Physical schema conversion strategy (D1). 53
4.4 Example of COBOL/SQL physical schema conversion. 54
4.5 Conceptual schema conversion strategy (D2) 55
4.6 Example of COBOL/SQL conceptual schema conversion. 57
4.7 Mapping-based data migration architecture. 57
4.8 A legacy COBOL code fragment.s 59
4.9 Wrapper-based migration architecture. 60
4.10 Fragment of Fig. 4.8 converted using the *Wrapper* strategy. 61
4.11 Fragment of Fig. 4.8 converted using the *Statement Rewriting* strategy. 63

4.12 Fragment of Fig. 4.8 converted using the *Logic Rewriting* strategy. . 65

5.1 A sample program and its corresponding SDG. 73
5.2 Sample program slice computed on the program of Figure 5.1. 74
5.3 Illustration of native, built-in, embedded, and call-based DMLs . . . 76
5.4 Methodology for slicing with embedded DML code 76
5.5 A COBOL/SQL code fragment 78
5.6 Methodology to slice programs with call-based DML 80
5.7 Calling program code 81
5.8 Data access module code 82
5.9 Successive steps to analyze the DAM call of Figure 5.7 83
5.10 A sample ASF equation for embedded SQL analysis 83
5.11 A sample ASF equation for IMS calls analysis 84

6.1 Two tables including implicit constructs 92
6.2 Two implicit constructs revealed by the analysis of Query 1 93
6.3 Seven capturing techniques for SQL statement executions. 103
6.4 Logging SQL operations by program instrumentation. 103
6.5 Illustration of API overloading. 104
6.6 Program adaptation for API overloading. 105
6.7 Reconstructing a fictitious update query using an update trigger . . 106
6.8 Tracing SQL query executions 110
6.9 Tracing SQL query executions in the presence of statement preparation111
6.10 Tracing SQL result extraction 112
6.11 A JDBC code fragment together with a corresponding execution trace115
6.12 An execution trace with output-input dependencies. 119
6.13 An execution trace with input-input dependencies. 119
6.14 Definition of two tracing tables. 122
6.15 Definition of intermediate views for each implicit foreign key. 124
6.16 Definition of intermediate tables for each implicit foreign key. 125
6.17 Counting foreign-key based joins between tables t_1 and t_2. 126
6.18 Counting foreign-key based output-input dependencies. 126

7.1 Example of a `SELECT` clause 134
7.2 Example of `FD` paragraph 134
7.3 Sample `ENVIRONMENT` division. 137
7.4 Sample `DATA` division transformation. 138
7.5 COBOL definition of the wrapper invocation arguments. 139
7.6 Replacement of a random `READ` statement with a wrapper invocation.140
7.7 SQL statements used to translate COBOL file handling primitives. . 141
7.8 One-to-one translation (D1) of a COBOL file into a relational table . 142
7.9 Definition of file `STUDENT`. 142
7.10 Example of a one-to-one schema conversion (D1). 143
7.11 A procedure that closes the current cursor. 146
7.12 `OPEN` translation for file `STUDENT`. 147

7.13 `OPEN OUTPUT` translation for file `STUDENT`. 147
7.14 `START` key usages VS SQL cursors. 148
7.15 `START` translation for file `STUDENT`. 148
7.16 `READ NEXT` translation for file `STUDENT`. 149
7.17 `READ KEY IS` translation for file `STUDENT`. 150
7.18 `WRITE` translation for file `STUDENT`. 150
7.19 `REWRITE` translation for file `STUDENT`. 151
7.20 `DELETE` translation for file `STUDENT`. 151
7.21 Example contents for file `STUDENT`. 152
7.22 Table `STUDENT` obtained from the file of Figure 7.21. 152
7.23 Tool architecture for program adaptation. 155
7.24 Case studies overview. 156

8.1 Migration architecture. 165
8.2 Example CODASYL to relational mapping. 166
8.3 CODASYL VS relational schema and instances. 169
8.4 Impact of `FIND` statements on the current record of a set type. 169
8.5 Example of legacy code transformation. 172
8.6 Tool architecture for program adaptation. 211
8.7 A code fragment of the wrapper generator. 213
8.8 Rewriting a `FIND NEXT WITHIN` *set* statement as a wrapper call. . . 213
8.9 Rewriting a `FIND NEXT WITHIN` *set* statement as a procedure call. . 214

9.1 Project 1: general architecture. 218
9.2 Project 2: refined methodology. 220
9.3 Project 2: Two-phase system migration. 223

10.1 Co-transformational approach. 233
10.2 Sample GER schema. 234
10.3 Approximate correspondences between data modification primitives. 236
10.4 Example LDA program allowing the creation of an ORDER. 238
10.5 Transformation of a compound attribute into an entity type. 239
10.6 Transformation of a one-to-many relationship type into a foreign key. 239
10.7 Graphical representation of a database co-transformation. 241
10.8 Propagation of the schema transformation of Figure 10.5. 242
10.9 Propagation of the schema transformation of Figure 10.6. 242
10.10Example of relational schema refactoring. 260
10.11Example CODASYL to relational schema conversion. 262
10.12Example design of a relational schema from a conceptual schema. . . 264
10.13Program of Figure 10.4 adapted to the schema of Figure 10.12. . . . 265
10.14Proof-of-concept tool support for database co-transformations. . . . 267
10.15GER constructs selection 269

A.1 Multiple possible reference keys for a single `READ NEXT` statement. . 299

List of Tables

3.1 Consistency relationships in data-intensive systems. 34
3.2 Semantic classification of GER schema modifications. 37

6.1 The SQL statements capturing techniques and their characteristics. . 109
6.2 Metrics about the SQL trace obtained. 123
6.3 Potential usage of implicit foreign keys by queries and scenarios. . . 127
6.4 Indications of the implicit foreign keys found in the SQL trace. . . . 128

8.1 SQL translation of `FIND NEXT WITHIN S` statements. 170

9.1 Project 1: Comparison of successive versions of the database schema. 219
9.2 Project 1: Legacy program transformation results. 219
9.3 Project 2: Legacy program refactoring results 221
9.4 Project 2: Comparison of successive versions of the database schema 222

List of 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
DAM	Data Access Module
DBCS	DataBase Control System
DBMS	DataBase Management System
DBRE	DataBase Reverse Engineering
DDL	Data Description Language
DML	Data Manipulation Language
DMS	Data Management System
DMS©	a Registered TradeMark of Semantic Designs Inc.
GER	Generic Entity-Relationship (model)
IDS/II	Integrated Data Store II
IMS	Information Management System
LDA	Langage de Description d'Algorithmes
PIM	Platform-Independent Model
PSM	Platform-Specific Model
SDG	System Dependency Graph
SDF	Syntax Definition Formalism
SPS	Source Physical Schema
SQL	Structured Query Language
3NF	Third Normal Form
TPS	Target Physical Schema
UWA	User Working Area

Chapter 1

Introduction

The best way to become acquainted with
a subject is to write a book about it.
– Benjamin Disraeli

1.1 Data-intensive software systems

Data-intensive software systems generally comprise a database (sometimes in the form of a set of files) and a collection of application programs in strong interaction with the former. They constitute critical assets in most entreprises, since they support business activities in all production and management domains. Data-intensive programs form most of the so-called legacy systems : they typically are one or more decade old, they are very large, heterogeneous and highly complex. Many of them *significantly resist modifications and change* (Brodie and Stonebraker, 1995) due to the use of ageing technologies and to inflexible architectures. Since they nevertheless are due to evolve, sophisticated techniques have been elaborated that allow programmers and developers to identify and understand the logic of the code fragments and of the data structures that are to be changed, despite the lack of precise and up to date documentation. Recovering the required knowledge and control of poorly documented software components is the main goal of software reverse engineering (Chikofsky and Cross, 1990).

1.2 The database component of software systems

It is interesting to learn how the communities devoted to the database and programming paradigms perceive the database component, both from the scientist and practitioner points of view. These views are quite different but complementary.

According to database experts, the database must be developed independently of the program needs. The goal of the database is to collect all the relevant data about a definite application domain (that part of the world concerned by the software system). The *Magna Carta* of a database is its *conceptual schema*, that identifies and describes the domain entities, their properties and their associations in a technology-independent way. The implementation of the database produces data structures that organize the data according to the data model of the chosen technology (the Database Management System or DBMS) but in strict conformance with the conceptual schema. The DBMS-dependent schema is called the *logical schema* of the database. In short, the conceptual schema expresses formally the intended semantics of the logical schema. Once the logical schema has been produced and coded in the Data Description Language (DDL) of the DBMS, application programs can be developed. In the final architecture, database experts perceive the database as the system's central component, around which the application programs are built.

For software engineering experts, system development largely ignores the database component. The latter appears as an encapsulated subsystem acting as a reliable and efficient data server. The database is an appropriate data container that ensures persistence, limited consistency, smooth concurrency and accident resistance. When external data are necessary, the programs invoke data extraction services from the DBMS through some sort of data manipulation language (DML). The same channel is used to send data to be stored in the database. The emphasis is less on the domain modeling aspect of the database than on convenience (such as storage transparency) and performance.

This thesis adopts a more balanced viewpoint. It argues that (1) both the database and the programs are important artefacts, (2) understanding what the programs are doing on the data may considerably help in understanding the database, and (3) database evolution methods should devote much more attention to the program adaptation problem.

1.3 On the complexity of data-intensive software systems

According to the modern description of pure software systems, a typical database can be perceived as a software sub-system made up of several thousands of classes, comprising dozens of thousands of attributes, and connected through several thousands of inter-class associations. Each class can collect several millions of persistent instances, so that a typical database forms a semantic network comprising billions of nodes and edges. These instances are shared, possibly simultaneously, by thousands of programs that read, create, delete and update several thousands times per second. Any of these programs can include hundreds of database statements of arbitrary complexity.

Since the database is supposed to include all the pertinent data about all the static object types of the application domain and since each program is designed to translate a business activity relying on these objects, it should not come as a surprise that data and processing aspects are tighly intertwined in application programs.

1.4 Evolution, maintenance and reverse engineering

Database schemas, software architecture and source code are supposed to be fully documented in order to make further maintenance and evolution easy and reliable. Unfortunately, development teams seldom have time to write and maintain a precise, complete and up to date documentation. Therefore, many complex software systems lack the documentation that would be necessary for maintenance and evolution. Faced with the necessity of frequently changing the program code and the database structure due to maintenance needs or to functional or technological evolution, developers perform local code analysis to try to understand how things work in these parts of the software and of the schema that should be modified.

The problem happens to be particularly complex for the database documentation due to prevalent developement practices. First of all, many databases have not been developed in a disciplined way, that is, from a preliminary conceptual schema. This was true for old systems, but loose empirical design approaches keep being widespread for modern databases due, notably, to time constraints, poor database education and the increasing use of object-oriented middleware that tends to consider the database as the mere implementation of program classes. Secondly, the logical (DBMS-dependent) schema, that is supposed to be derived from the conceptual schema and to translate all its semantics, generally misses several conceptual constructs. This is due to several reasons, among others the poor expressive power of DBMS models and the lazziness, awkwardness or illiteracy of some programmers (Blaha and Premerlani, 1995). From all this, it results that the logical model often is incomplete and that the DDL code that expresses the DBMS schema in physical constructs ignores important structures and properties of the data. The missing constructs are called *implicit*, in contrast with the *explicit constructs* that are declared in the DDL of the DBMS. Several field experiments and projects have shown that as much as half of the semantics of the data structures is implicit. Therefore, merely parsing the DDL code of the database, or, equivalently, extracting the physical schema from the system tables, sometimes provides barely half the actual data structures and integrity constraints.

Recovering the implicit constructs is a prerequisite to maintenance and evolution of both the database and the programs. It also proves fairly difficult. Indeed, it relies on such complex techniques as data mining, source code analysis, graphical interface analysis and program trace analysis.

1.5 System reengineering and migration

As defined by Chikofsky and Cross (1990), *reengineering, also known as [...] renovation [...], is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some more form of forward engineering or restructuring.* Migration is a variant of reengineering in which the transformation is driven by a major technology change.

A large part of this thesis addresses a particular case of system migration, namely database platform migration. Replacing a DBMS with another one should, in an ideal world, only impact the database component of the information system. Unfortunately, the database most often has a deep influence on other components, such as the application programs. Two reasons can be identified. First, the programs invoke data management services through an API that generally relies on complex and highly specific protocols. Changing the DBMS, and therefore its protocols, involves the rewriting of the invocation code sections. Second, the database schema is the technical translation of its conceptual schema through a set of rules that is dependent on the DBMS data model. Porting the database to another DBMS, and therefore to another data model, generally requires another set of rules, that produces a significantly different database schema. Consequently, the code of the programs often has to be adapted to this new schema. Clearly, the renovation of an information system by replacing an obsolete DBMS with a modern data management system leads to non trivial database (schemas and data) and programs modifications.

1.6 Goals of the Thesis

The general goal of this thesis is to contribute to the automated support of data-intensive systems evolution. More particularly, the thesis aims at proposing methodologies, techniques and tools for :

1. analyzing legacy data-intensive programs in support to database reverse engineering;

2. adapting legacy data-intensive programs to database evolutions in general, and to database platform migration in particular.

1.7 Research questions

This thesis research is driven by the following research questions:

> **RQ1** : *Can automated program analysis techniques help to recover implicit knowledge on the structure and constraints of a database?*
>
> **RQ2** : *What are the possible strategies for migrating a legacy data-intensive system towards a modern database platform? How do they compare?*
>
> **RQ3** : *Is it possible to automatically adapt large legacy systems to the migration of their underlying database?*
>
> **RQ4** : *How to preserve the consistency between an evolving database schema and associated queries?*

1.8 Outline of the Thesis

The remaining of this thesis is composed of six parts and ten chapters, as depicted in Figure 1.1.
Part I regroups the introductory chapters of the thesis, related to its general research domain.

- **Chapter 2** provides a short introduction to the main basic concepts used in the thesis.
- **Chapter 3** presents a comprehensive reference framework for the evolution of data-intensive systems. It identifies and describes the main consistency relationships that hold in data-intensive applications. Based on those relationships, it proposes a classification of database evolution scenarios. This classification is then used to revisit the thesis goals and to better position its contributions.

Part II is dedicated to a particular case of data-intensive system evolution, namely system migration.

- **Chapter 4** discusses existing work on software-intensive system migration, and develops a two-dimensional reference framework for database platform migration. This framework identifies and compares six representative system migration strategies.

Part III elaborates on the use of program analysis techniques for supporting the typical initial phase of any database evolution, namely database reverse engineering.

- **Chapter 5** concentrates on static program analysis. It presents a tool-supported approach to extracting dataflow dependencies from database queries. It shows that these techniques may contribute, among others, to the recovery of implicit knowledge on the database structures and constraints.
- **Chapter 6** explores the use of dynamic program analysis techniques for reverse engineering relational databases. The static analysis techniques presented in Chapter 5 may indeed fall short in the presence of automatically generated SQL queries. The chapter therefore presents and compares possible techniques for (1) capturing the SQL query executions at run time and (2) extracting implicit schema constructs from SQL query execution traces.

Part IV presents DMS-specific methods and tools for the migration of legacy data-intensive systems. In this part, we assume that a legacy database has been migrated towards a modern platform, and we show how to automatically adapt the legacy programs accordingly.

- **Chapter 7** is dedicated to the migration of standard files towards a relational database platform. It provides systematic program conversion rules allowing to translate COBOL file handling primitives into equivalent SQL queries.

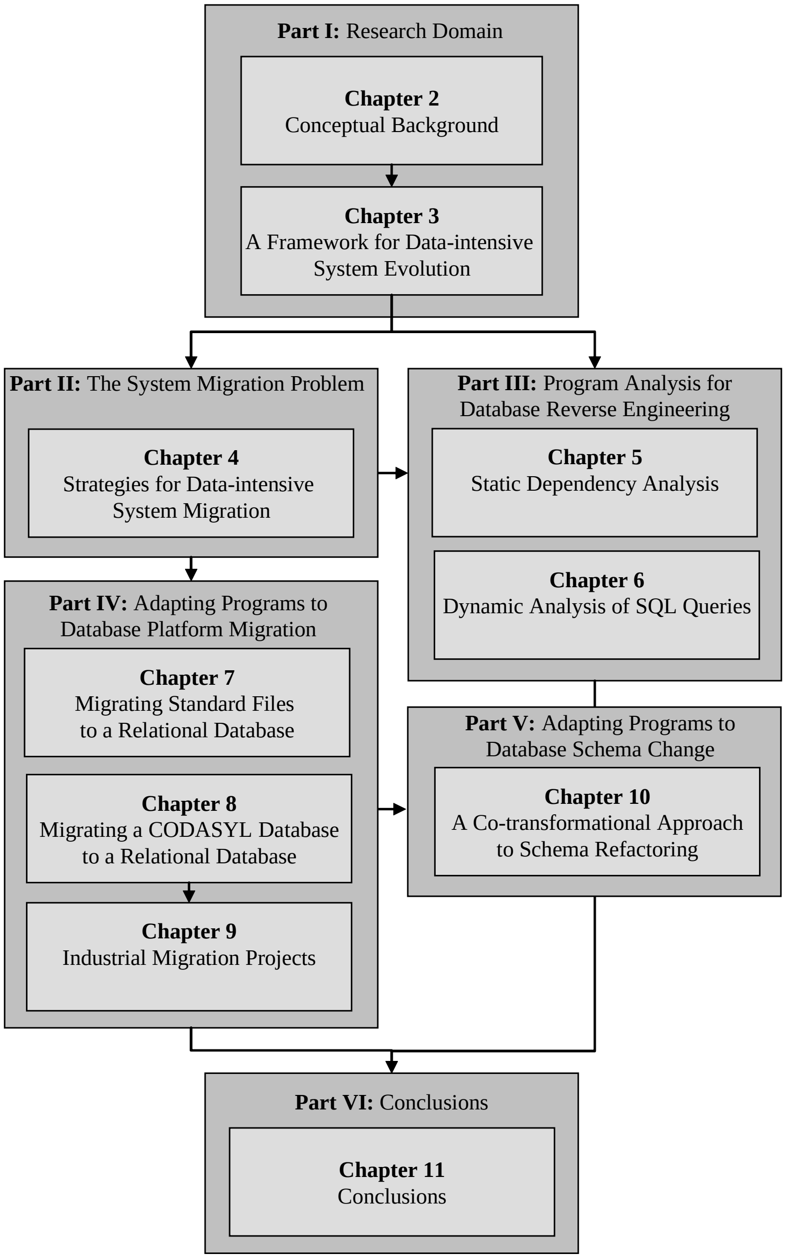

Part I: Research Domain
Chapter 2
Conceptual Background
Chapter 3
A Framework for Data-intensive System Evolution
Part II: The System Migration Problem
Chapter 4
Strategies for Data-intensive System Migration
Part III: Program Analysis for Database Reverse Engineering
Chapter 5
Static Dependency Analysis
Chapter 6
Dynamic Analysis of SQL Queries
Part IV: Adapting Programs to Database Platform Migration
Chapter 7
Migrating Standard Files to a Relational Database
Chapter 8
Migrating a CODASYL Database to a Relational Database
Chapter 9
Industrial Migration Projects
Part V: Adapting Programs to Database Schema Change
Chapter 10
A Co-transformational Approach to Schema Refactoring
Part VI: Conclusions
Chapter 11
Conclusions

Figure 1.1: General structure of the thesis.

- **Chapter 8** addresses another popular, but much more challenging migration scenario: the conversion of a CODASYL database towards a relational platform. It presents a tool-supported approach for simulating CODASYL data manipulation statements on top of a relational database.

- **Chapter 9** discusses the application of our migration approach and tools (presented in Chapter 8) in the context of real-size industrial migration projects.

Part V is devoted to another particular database evolution scenario, namely database schema change. More precisely, it aims at supporting the co-evolution of database schema and programs.

- **Chapter 10** presents a co-transformational approach to the propagation of database schema changes on application programs. This approach consists in systematically associating abstract program transformation rules to a set of semantics-preserving schema transformations. The chapter then shows that such abstract propagation rules may be reused in other database engineering contexts such as database migration and database design.

Part VI draws the general conclusions of this thesis and anticipates future work.

- **Chapter 11** summarizes and evaluates the contributions of the thesis with respect to the above research questions. It also discusses the lessons we learned from this work and identifies avenues for future research in the field.

Figure 1.1 depicts the general structure of the thesis and identifies the main dependencies between the succesive parts and chapters.

1.9 Publications

Most of the research results presented in this thesis have been published as book chapters or in international conferences and workshops. Below, we provide a list of selected peer-reviewed publications, in chronological order:

1. Anthony Cleve and Jean-Luc Hainaut. Co-transformations in database applications evolution. In Ralf Lämmel, João Saraiva and Joost Visser, editors, Post-proceedings of GTTSE 2005, Generative and Transformation Techniques in Software Engineering, Vol. 4143 of *Lecture Notes in Computer Science*, pages 409–421. Springer, 2006.

2. Anthony Cleve. Automating program conversion in database reengineering - a wrapper-based approach. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 323–326., IEEE Computer Society, 2006.

3. Anthony Cleve, Jean Henrard, and Jean-Luc Hainaut. Data reverse engineering using system dependency graphs. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06)*, pages 157–166. IEEE Computer Society, 2006.

4. Jean Henrard, Didier Roland, Anthony Cleve, and Jean-Luc Hainaut. An industrial experience report on legacy data-intensive system migration. In *Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07)*, pages 473–476. IEEE Computer Society, 2007.

5. Jean-Luc Hainaut, Anthony Cleve, Jean Henrard, and Jean-Marc Hick. Migration of legacy information systems. In Tom Mens and Serge Demeyer, editors, *Software Evolution*, pages 105–138. Springer, 2008.

6. Anthony Cleve, Jean Henrard, Didier Roland, and Jean-Luc Hainaut. Wrapper-based system evolution - application to CODASYL to relational migration. In *Proceedings of the 12th European Conference in Software Maintenance and Reengineering (CSMR'08)*, pages 13–22. IEEE Computer Society, 2008.

7. Anthony Cleve and Jean-Luc Hainaut. Dynamic analysis of SQL statements for data-intensive applications reverse engineering. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 192–196. IEEE Computer Society, 2008.

8. Jean Henrard, Didier Roland, Anthony Cleve, and Jean-Luc Hainaut. Large-scale data reengineering: Return from experience. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08)*, pages 305–308. IEEE Computer Society, 2008.

Part I

Research Domain

Chapter 2

Conceptual Background

A journey of a thousand miles begins with a single step.
– Confucius

As its title indicates, this chapter aims at providing the reader with a conceptual background, by introducing the main concepts manipulated in the thesis. The following sections briefly elaborate on the *database design* and *database reverse engineering* processes, the *GER model* and the *transformational approach* to data-intensive systems engineering.

2.1 Database design

The process of designing and implementing a database that has to meet specific user requirements has been described extensively in the literature (Batini et al., 1992) and has been available for several decades in standard methodologies and CASE tools. As shown in Figure 2.1, database design is typically made up of four main subprocesses:

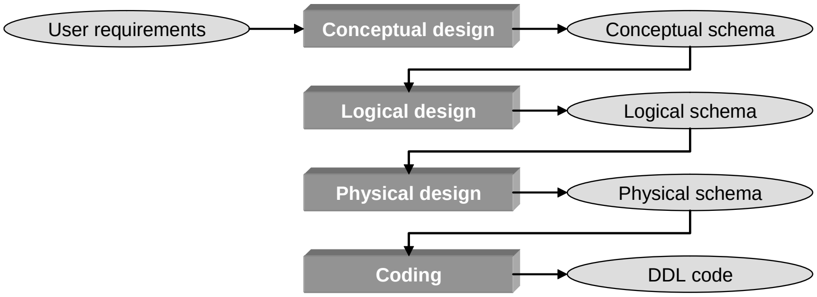


Figure 2.1: Standard database design processes.

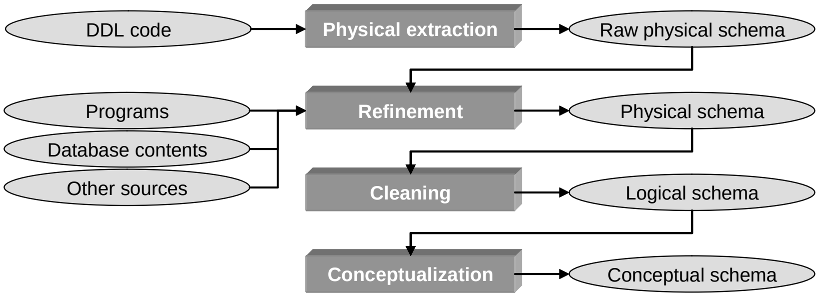


Figure 2.2: Standard database reverse engineering processes.

(1) *Conceptual design* is intended to translate user requirements into a *conceptual schema* that identifies and describes the domain entities, their properties and their associations in a technology-independent way. This abstract specification of the future database collects all the information structures and constraints of interest.

(2) *Logical design* produces an operational *logical schema* that translates the constructs of the conceptual schema according to a specific technology family without loss of semantics.

(3) *Physical design* augments the logical schema with performance-oriented constructs and parameters, such as indexes, buffer management strategies or lock management policies.

(4) *Coding* translates the physical schema (and some other artefacts) into the DDL (*Data Definition Language*) code of the database management system. Structural DDL declaration code as well as components such as checks, triggers and stored procedures are generated to code the information structures and constraints of the physical schema.

2.2 Data(base) reverse engineering

Chikofsky (1996) define data reverse engineering as "*a collection of methods and tools to help an organization determine the structure, function, and meaning of its data*". Database Reverse Engineering (DBRE) is *the process through which the logical and conceptual schemas of a legacy database, or of a set of files, are reconstructed from various information sources such as DDL code, data dictionary contents, database contents or the source code of applications that use the database.* (Hainaut et al., 2009).

As depicted in Figure 2.2, DBRE typically comprises the following four subprocesses:

(1) *Physical extraction* consists in parsing the DDL code in order to extract the raw physical schema of the database.

(2) *Refinement* enriches the raw physical schema with additional structures and constraints elicited through the analysis of the application programs and other sources.

(3) *Cleaning* removes the physical constructs (such as indexes) for producing the logical schema.

(4) *Conceptualization* aims at deriving the conceptual schema that the logical schema obtained implements.

In this thesis, we will particularly contribute to the refinement process, by proposing program analysis techniques allowing to discover implicit schema constructs.

2.3 The Generic Entity-Relationship model

The Generic Entity-Relationship model (Hainaut, 1989), GER for short, is an extended Entity-Relationship model including, among others, the concepts of *schema*, *entity type*, *domain*, *attribute*, *relationship type*, *key*, as well as various constraints. The GER model encompasses the three main levels of abstractions for database schemas, namely conceptual, logical and physical. It also serves as a generic pivot model between the major database paradigms including ER, relational, object-oriented, object-relational, files structures, network, hierarchical, XML.

Below, we further present and illustrate the GER model according to the three levels of abstraction.

2.3.1 Conceptual schemas

A conceptual schema mainly specifies *entity types*, *relationship types* and *attributes*. Entity types represent the main concepts of the application domain. They can be organized into *is-a* hierarchies, organizing supertypes and subtypes. An is-a hierarchy can be total and/or disjoint. Total (T) means that a supertype must be specialized in at least one subtype. Disjoint (D) means that a supertype can be specialized in at most one subtype. A partition (noted P) corresponds to an is-a hierarchy that is both total and disjoint.

Relationship types represent relationships between entity types. A relationship type has two or more roles. A role has a cardinality constraint [i-j], that specifies in how many relationships an entity can appear with this role. A relationship type with exactly two roles is called *binary*, while a relationship type with more than two roles is generally called *n-ary*.

Entity types and relationship types can have attributes, which can be either atomic or compound. A compound attribute is an attribute that is made of at least one sub-level attribute (atomic or compound).

Attributes are also characterized with a cardinality constraint [i-j] (0 <= i <= j) specifying how many values can be associated with a parent instance. The minimum cardinality (i) states how many attribute values *must* be associated, while the maximum cardinality (j) corresponds to maximum number of values that *can* be associated.

By default, the cardinality constraint of an attribute is [1-1]. A *mandatory* (resp. *optional*) attribute is an attribute for which the minimum cardinality is equal 1 (resp. 0). A *single-valued* (resp. *multivalued*) attribute is an attribute for which the maximum cardinality is 1 (resp. >1).

Entity types and relationship types may be given possibly complex constraints. Those constraints are expressed through the concept of *group.* A group is a logical set of elements (attributes, roles and/or other groups) attached to a parent object (entity type, relationship type or compound attribute). It is used, among others, to represent such constraints as uniqueness, exclusion and coexistence:

- *primary identifier* (`id`) The elements of the group form the main identifier of the parent object. A parent object can have at most one primary identifier. All components of an `id` group must be mandatory.
- *secondary identifier* (`id'`) The elements of the group make up a secondary identifier of the parent object. A parent object can have several secondary identifiers.
- *coexistence* (`coex`): Either all elements of the group have a value or none for any instance of the parent object.
- *exclusive* (`excl`): Among the elements of the group, at most one can have a value for any instance of the parent object.
- *at-least-one* (`at-lst-1`): Among the elements of the group, at least one must have a value for any instance of the parent object.
- *exactly-one* (`exact-1`) Among the elements of the group, one and only one can have a value for any instance of the parent object. This corresponds to the combination of the *exclusive* and *at-least-one* constraints.

Figure 2.3 depicts an example of GER conceptual schema. This schema includes entity types `PERSON`, `CUSTOMER`, `SUPPLIER`, `ORDER` and `PRODUCT`. `PERSON` has two disjoint subtypes, `CUSTOMER` and `SUPPLIER`. Relationship type `from` is binary while `detail` is ternary. Each `ORDER` entity appears in exactly one `from` relationship (cardinality [1-1]), and in at least one `detail` relationship (cardinality [1-N]). For entity type `PERSON`, attribute `Name` is atomic, single-valued and mandatory. `Address` is a compound attribute. Its component `Num` is atomic, single-valued and optional (cardinality [0-1]). `Phone` is multivalued and optional: there are from 0 to 5 values per entity. {`PID`} is the identifier of `PERSON`. The identifier of `ORDER` is made of external entity type `from.CUSTOMER` and of local attribute `ONum`. There cannot exist more than one `detail` relationship with the same `ORDER` and `PRODUCT` entites.

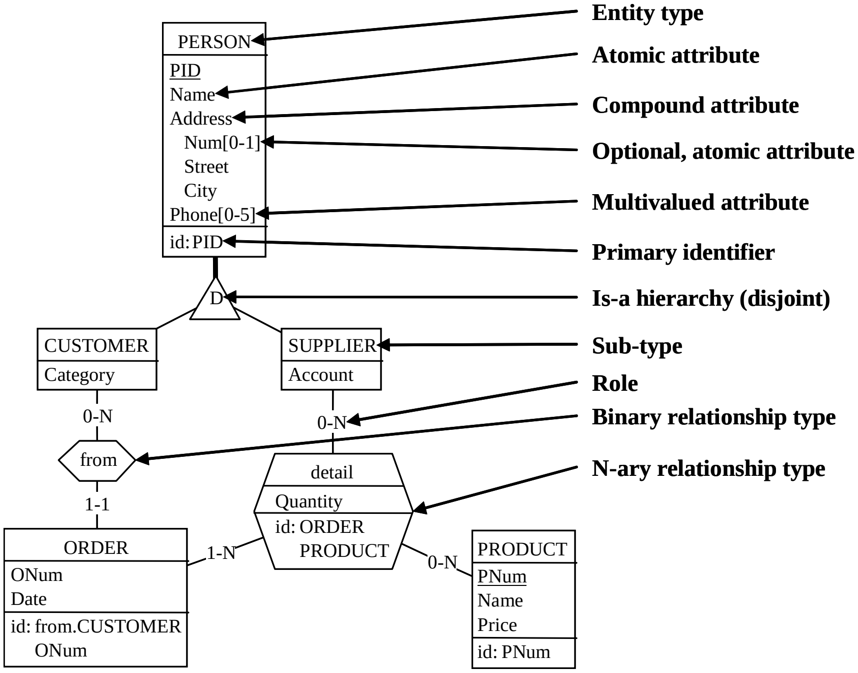


Figure 2.3: Sample GER conceptual schema.

2.3.2 Logical schemas

A logical schema is a platform-dependent data structure definition, that must comply with a given data model. The most commonly used families of models include the relational model, the network model (CODASYL DBTG), the hierarchical model (IMS), the standard file model (COBOL, C, RPG, BASIC), the shallow model (TOTAL, IMAGE), the XML model, the object-oriented model and the object-relational model.

A logical schema basically uses the same schema constructs as the ones presented in Section 2.3.1 for conceptual schemas. Depending on the logical model the same schema constructs are called differently. For instance, a GER entity type (resp. attribute) is called a *table* (resp. *column*) in the relational terminology, and is called *record type* (resp. *field*) in a CODASYL schema.

Each logical model has its own set of *allowed* schema constructs. For instance, a relational schema may not comprise relationship types, compound attributes, multivalued attributes and *is-a* hierarchies. Such illegal constructs must be expressed by equivalent constructs (if any) of the target logical model. It can happen that some fragments of a conceptual schema cannot be fully translated into the logical schema.

In addition to the ones described above, new schema constructs may also appear at the logical level:

- *Referential constraint* (`ref`): An inter-group constraint between an *origin group* (`ref` group) and a *target group* that states that each instance of the origin group must correspond to an instance of the target group. The target group must represent an identifier (`id` or `id'`). A referential constraint is called a *foreign key* in the relational model.

- *Inclusion constraint* (`incl`): An inter-group constraint where each instance of the origin group must be an instance of the target group. Here, the target group does not need to be an identifier (generalization of the referential constraint).

- *Equality constraint* (`equ`): A referential constraint between an origin group r and a target group i, combined with an inclusion constraint defined from the i to r.

- *Typed multivalued attribute*: In a conceptual schema, multivalued attributes represent sets of values, i.e. unstructured collections of distinct values. At the logical level, we can distinguish six possible implementations of a multivalued attribute:

  - *Set*: unstructured collection of distinct elements (default).
  - *Bag*: unstructured collection of (not necessarily distinct) elements.
  - *Unique list*: sequenced collection of distinct elements.
  - *List*: sequenced collection of (not necessarily distinct) elements.

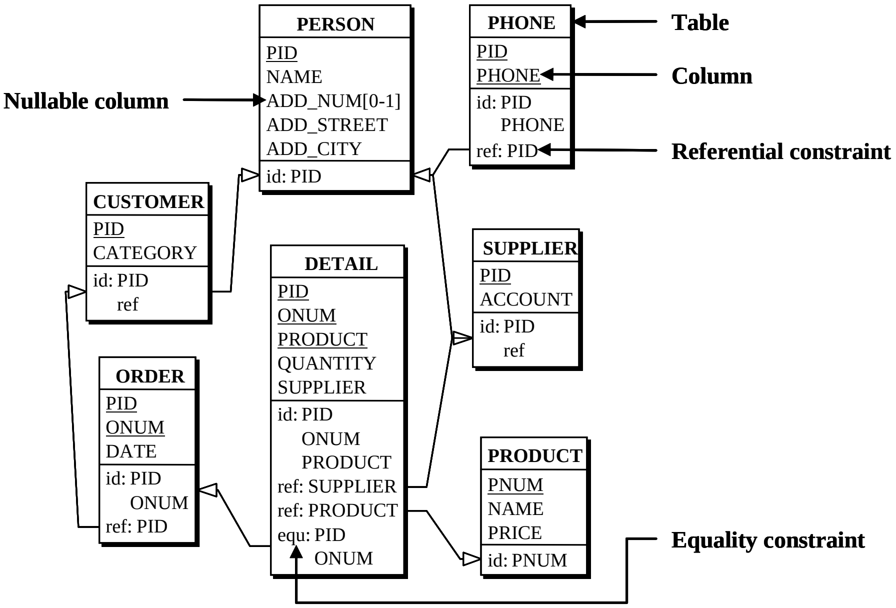


Figure 2.4: Sample GER logical schema, approximate relational translation of the conceptual schema of Figure 2.3.

- *Unique array*: indexed sequence of cells that can each contain an element. The elements are distinct.
- *Array*: indexed sequence of cells that can each contain an element.

An example fragment of logical schema is given in Figure 2.4. This relational schema corresponds to an approximate translation of the conceptual schema depicted in Figure 2.3.1. The schema defines seven tables. Table PERSON has mandatory columns (PID, NAME, ADD_STREET and ADD_CITY) and one optional (nullable) column, ADD_NUM. Its primary identifier is {PID}. Column {PID} of ORDER is a foreign key to CUSTOMER (targeting its primary id). The group {PID, ONUM} of DETAIL is a multicomponent foreign key. In addition, there is an inclusion constraint from {PID, ONUM} of ORDER to {PID, ONUM} of DETAIL. Combining these two constraints translates into an equality constraint (equ). {PID} of CUSTOMER is both a primary id and a foreign key to PERSON.

2.3.3 Physical schemas

A physical schema is a logical schema enriched with all the information needed to implement efficiently the database on top of a given data management system. This includes DMS-dependent technical specifications such as indexes, physical device and site assignment, page size, file size, buffer management or access right policies. Due to their large variety, it is not easy to propose a general model covering all possible physical constructs. In the scope of this thesis, we will make use of the two following concepts:

- *record collection*, which is an abstraction of file, data set, tablespace, dbspace and any record repository in which data are permanently stored;
- *access key* (`acc`), which represents any path providing a fast and selective access to records that satisfy a definite criterion; indexes, indexed set (DBTG), access path, hash files, inverted files, indexed sequential organizations all are concrete instances of the concept of access key.

Figure 2.5 depicts a physical GER schema that derives from the logical schema of Figure 2.4. This schema is made up of seven tables and three collections. Collection `PERS_FILE` stores instances of tables `PERSON`, `CUSTOMER`, `SUPPLIER` and `PHONE`. The primary identifiers and some foreign keys are supported by an access key (groups denoted by `acc`). Access keys are also associated with two regular columns (`PERSON.NAME` and `PRODUCT.NAME`).

2.4 The transformational approach

Any process that consists in deriving artefacts from other artefacts relies on such techniques as renaming, translating, restructuring, replacing, refining and abstracting, which basically are *transformations*. Most database engineering processes can be formalized as chains of elementary schema and data transformations that preserve some of their aspects, such as its information contents (Hainaut, 2006). Information system evolution, and more particularly system migration as defined in this thesis, consists of the transformation of the database and of its programs into a new system comprising the modified database and the adapted programs. As far as programs are concerned, the transformations must preserve the behaviour of the interface with the database management system, although both the syntax of this interface and/or the underlying data structure may undergo some changes.

2.4.1 Schema transformation

Roughly speaking, an elementary schema transformation consists in deriving a target schema S' from a source schema S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty). C (resp. C') is empty when the transformation consists in adding (resp. removing) a construct. Adding an attribute to an entity type, replacing a relationship type by an equivalent entity type or by a foreign key and replacing an attribute by an entity type (Figure 2.7) are some examples of schema transformations.

More formally, a transformation Σ is defined, as shown in Figure 2.6, as a couple of mappings $\langle T, t\rangle$> such that, $C' = T(C)$ and $c' = t(c)$, where c is any instance

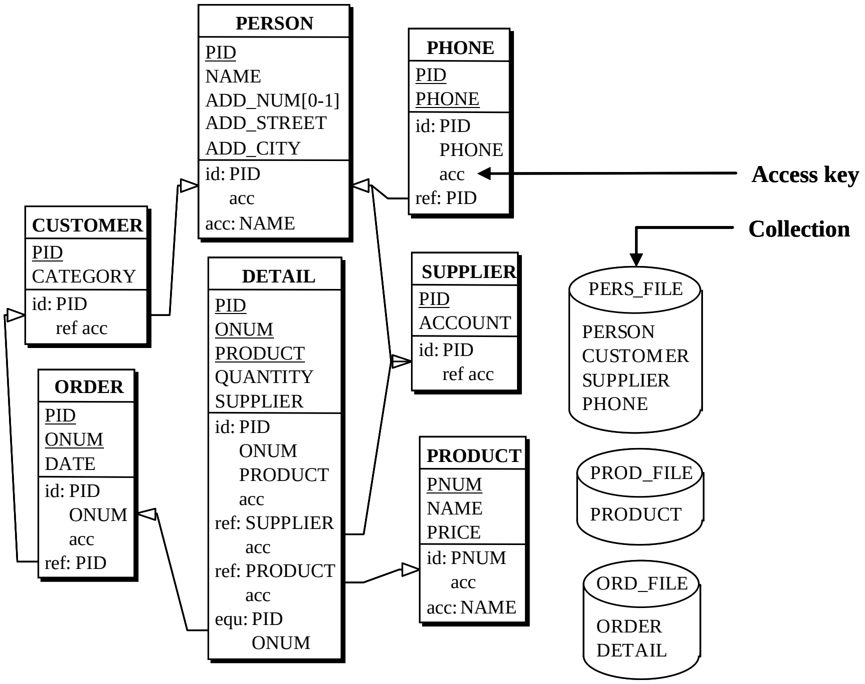


Figure 2.5: Sample GER physical schema.

$$\begin{tikzcd}
C \arrow[r, "T"] & C' = T(C) \\
c \arrow[u, "\textit{instance-of}"] \arrow[r, "t"] & c' = t(c) \arrow[u, "\textit{instance-of}"']
\end{tikzcd}$$

Figure 2.6: Schema transformation defined as a couple of structural and instance mappings.

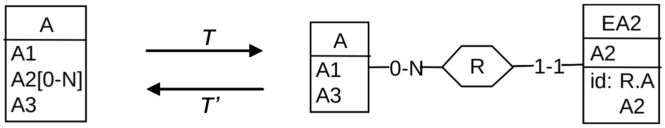


Figure 2.7: Pattern-based representation of the structural mapping of *ATTRIBUTE-to-ET* transformation that replaces a multivalued attribute (`A2`) by an entity type (`EA2`) and a relationship type (`R`).

of C and c' the corresponding instance of C'. *Structural mapping* T is a rewriting rule that specifies how to modify the schema while *instance mapping* t states how to compute the instance set of C' from the instances of C.

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

Any transformation Σ can be given an inverse transformation $\Sigma' = \langle T', t' \rangle$ such that $T'(T(C)) = C$. If, in addition, we also have: $t'(t(c)) = c$, then Σ (and Σ') are called semantics-preserving[1]. Figure 2.7 shows a popular way to convert an attribute into an entity type (structural mapping T), and back (structural mapping T'). The instance mapping, that is not shown, would describe how each instance of source attribute `A2` is converted into an entity type `EA2` and a relationship type `R`.

Practically, the application of a transformation will be specified by its signature, that identifies the source objects and provides the names of the new target objects. For example, the signatures of the transformations of Figure 2.7 are:

$$\begin{array}{lcll}
T & : & (\texttt{EA2}, \texttt{R}) & \leftarrow \textit{ATTRIBUTE-to-ET}(\texttt{A}, \texttt{A2}) \\
T' & : & (\texttt{A2}) & \leftarrow \textit{ET-to-ATTRIBUTE}(\texttt{EA2})
\end{array}$$

[1] The concept of semantics (or information contents) preservation is more complex, but this definition is sufficient in this context. A more comprehensive definition can be found in Hainaut (2006).

Transformations such as those in Figure 2.7 include names (`A`, `A1`, `R`, `EA2`, etc.) that actually are variable names. Substituting names of objects of an actual schema for these abstract names provides fully or partially instantiated transformations. For example:

$$(\texttt{'PHONE'}, \texttt{'has'}) \leftarrow \textit{ATTRIBUTE-to-ET}(\texttt{'CUSTOMER',' Phone'})$$

specifies the transformation of attribute `Phone` of entity type `CUSTOMER`. Similarly,

$$(\texttt{EA2}, \texttt{R}) \leftarrow \textit{ATTRIBUTE-to-ET}(\texttt{'CUSTOMER'}, \texttt{A2})$$

specifies the family of transformations of any attribute of `CUSTOMER` entity type.

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

2.4.2 Compound schema transformation

A compound transformation $\Sigma = \Sigma_2 \circ \Sigma_1$ is obtained by applying Σ_2 on the database (schema and data) that results from the application of Σ_1 (Hainaut, 1996). Most complex database engineering processes, particularly database design and reverse engineering, can be modelled as compound semantics-preserving transformations. For instance, transformation *CS-to-PS* referred to here above actually is a compound transformation, since it comprises logical design, that transforms a conceptual schema into a logical schema, followed by physical design, that transforms the logical schema into a physical schema (Batini et al., 1992). So, the database design process can be modelled by transformation $CS\text{-}to\text{-}PS = LS\text{-}to\text{-}PS \circ CS\text{-}to\text{-}LS$, while the reverse engineering process is modelled by $PS\text{-}to\text{-}CS = LS\text{-}to\text{-}CS \circ PS\text{-}to\text{-}LS$.

2.4.3 Transformation history and schema mapping

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

In particular, it formally and completely defines the mapping between a source schema and its target counterpart when the latter was produced by means of a transformational process. Indeed, the chain of transformations that originates from any definite source object precisely designates the resulting objects in the target schema, as well as the way they were produced. However, the history approach to mapping specification has proved complex, essentially for three reasons (Hainaut et al., 1996b). First, a history includes information that is useless for schema migration. In particular, the signatures often include additional information for undoing and inverting transformations. Second, making histories evolve consistently over time is far from trivial. Third, real histories are not linear, due to the exploratory nature of engineering processes.

Therefore, simpler mappings are often preferred[2], even though they are less powerful. For instance, Hick and Hainaut (2006) proposed the use of the following lighweight technique based on stamp propagation. Each source object receives a unique stamp that is propagated to all objects resulting from the successive transformations. When comparing the source and target schemas, the objects that have the same stamp exhibit a pattern that uniquely identifies the transformation that was applied on the source object. This approach is valid provided that (1) only a limited set of transformations is used and (2) the transformation chain from each source object is short (one or two operations). Fortunately, these conditions are almost always met in real database design.

2.4.4 Program transformation

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

Automated program transformations form a sound basis for application program adaptation in the context of data-intensive systems maintenance and evolution. Indeed, the size of such systems often calls for automated tool support for program modification, the latter being tedious and error-prone when performed manually.

2.4.5 Program analysis and transformation technology

There exist several generic language technology (GLT) systems that support automatic program analysis and transformation. Among such systems, let us mention the ASF+SDF Meta-Environment (van den Brand et al., 2001), Stratego/XT (Bravenboer et al., 2008), TXL (Cordy et al., 2002), DMS (Baxter et al., 2004) and the Synthesizer generator (Reps and Teitelbaum, 1984).

In this research, we mainly used the ASF+SDF Meta-Environment to carry out the automated analysis and renovation of large-scale COBOL systems. Below, we briefly justify this choice with respect to several aspects of our application domain.

[2] This will be the case in this thesis

Availability The ASF+SDF Meta-Environment is available for free (van den Brand et al., 2003), as opposed to DMS (Baxter et al., 2004) and the Synthesizer generator (Reps and Teitelbaum, 1984), which are commercial products.

Suitability for large-scale software analysis and renovation The ASF+SDF Meta-Environment has been successfully used to support software renovation tasks of various nature and for various languages (van den Brand et al., 2007), including COBOL system prettyprinting (van den Brand et al., 2006) and restructuring (Veerman, 2004), code smell detection in Java (van Emden and Moonen, 2002) and renovation of legacy C code towards aspect-oriented programming (Bruntink, 2008a). More importantly, several industrial applications have shown its suitability for the automated maintenance of *large-scale* software systems (van den Brand et al., 1996; Bruntink, 2008b) in general, and legacy COBOL systems in particular (Veerman, 2007).

Grammar modularity The Syntax Definition Formalism (SDF) of the ASF+SDF Meta-Environment allows the definition of a language grammar in a modular way. This feature is very important, especially in the case of the COBOL language, for which various dialects exist. Adapting a program analysis or transformation tool to a new dialect is made easier when only a few syntactic modules have to be changed.

Traversal functions The ASF+SDF Meta-Environment provides the user with *traversal functions* (van Den Brand et al., 2003), that constitute a very convenient support for generic parse tree traversal. Those functions allow to specify the analysis and rewriting of complex parse trees in a very consise way, thereby focusing only on the nodes of interest. We made an intensive usage of such traversal functions when building our analysis and transformation tools, that mainly target the database manipulation statements occuring in legacy programs.

Layout and comment preservation The ASF+SDF Meta-Environment preserves the layout and comments of the source code fragments that are not rewritten (van den Brand and Vinju, 2000). In our work, this feature is also essential, for the several reasons. First, standard prettyprinters are usually not acceptable in an industrial context. Second, source code comments are invaluable for the maintenance teams and thus cannot be lost. Last, in some programming languages like COBOL, whitespaces are semantically relevant.

Roadmap

This chapter has briefly presented the conceptual background of the thesis. In the next chapter (Chapter 3), we really embark on the thesis topic by presenting a comprehensive reference framework for the evolution of data-intensive systems. This framework is then used to clarify the goals of the thesis.

Chapter 3

A Framework for Data-Intensive System Evolution

Science is the systematic classification of experience.
– George Henry Lewes

This chapter aims at providing a comprehensive reference framework for the evolution of data-intensive systems. The chapter starts with an in-depth analysis of the consistency relationships that hold between the main artefacts of data-intensive systems. It proposes a classification of database evolution scenarios and identifies the typical processes involved in database evolution. Finally, it makes use of the framework for clarifying the scope, contributions and outline of the thesis in terms of database evolution scenarios and processes.

3.1 The nature of consistency relationships

Before we can propose a classification of evolution scenarios for data-intensive systems, it is essential to identify the various dependency relationships that hold between the system artefacts and, more importantly, to understand their nature. The main inter-dependent artefacts, that we will consider in this thesis, are the following:

- the **database schemas**, each at their underlying level of abstraction: conceptual, logical and physical;
- the **DDL code**, translating the physical database schema;
- the **data instances**, i.e., the current state of the database contents;
- the **application programs**, manipulating the database through a given data manipulation language (DML);

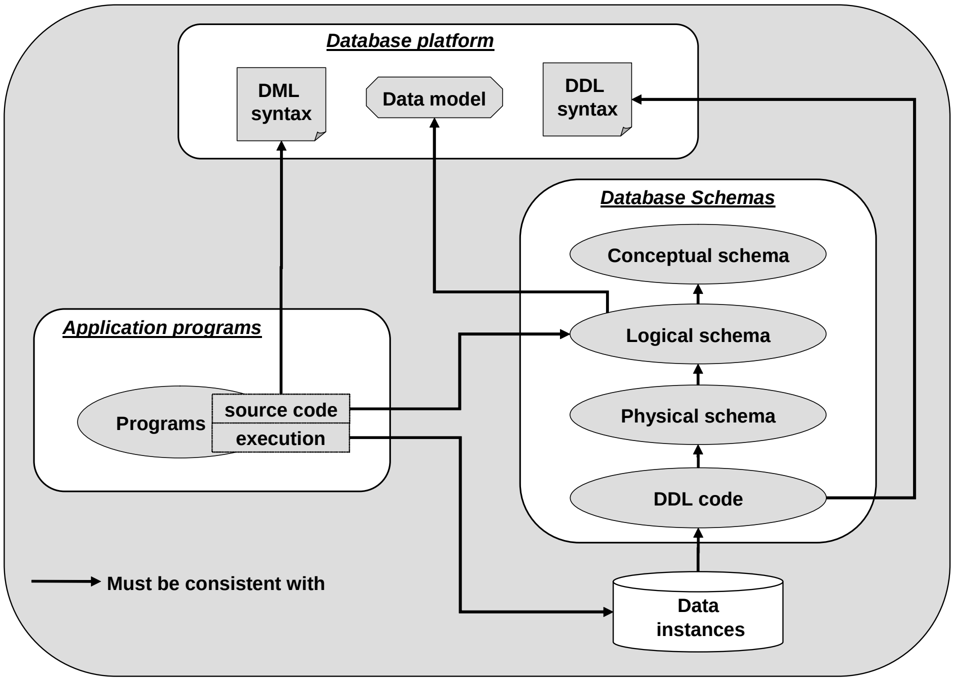


Figure 3.1: Consistency relationships in data-intensive applications.

- the **DDL syntax**, expressing the platform-specific way of defining the database structure;
- the **DML syntax**, i.e., the platform-specific grammar of the data manipulation language.

In Figure 3.1, we identify the major consistency relationships that must hold between those artefacts. Although the consistency relationships often are symmetric, the direction of the arrows indicates the most intuitive way of interpreting them. Each arrow can be read as '*must be consistent with*'. Below we further describe and illustrate the nature of each relationship identified.

Logical schema VS conceptual schema

According to the MDE[1] terminology, the logical schema (LS) is a platform-specific model (PSM) derived from the conceptual schema (CS), which is a platform-independent model (PIM). The relationship between both schemas can be modeled by a chain of (semantics-preserving) schema transformations. The consistency between LS and CS is guaranteed if $\exists\ \Sigma_{C2L} : LS = \Sigma_{C2L}(CS)$, with Σ_{C2L} a chain of semantics-preserving transformations. Figure 3.2 illustrates the relationship between the conceptual and logical schemas in the case of semantics-preserving logical design. In this example, all the constructs belonging to the conceptual schema have been translated into equivalent constructs in the logical schema. In practice, however, it may happen that non-semantics-preserving transformations are applied during the logical design process. This typically occurs when the target logical model does not allow certain schema constructs. For instance, the COBOL model does not include relationship types, the latter should be discarded and replaced with so-called *implicit* (or *undeclared*) foreign keys. The same holds for early versions of MySQL, which do not support foreign keys. In both cases, the referential constraints must be managed by the application programs themselves. Figure 3.3 illustrates such a situation, where the relationship types occuring in the conceptual schema have not been fully translated into corresponding foreign keys in the logical schema.

In summary, in order to qualify the nature of the consistency between a logical schema LS and a conceptual schema CS, we will say that a LS must be *semantically compatible* (instead of equivalent) with CS. This means that we allow some information loss during logical design, but we impose that every construct belonging to LS derives from construct(s) of CS.

Physical schema VS logical schema

The consistency relationship that must hold between the physical schema (PS) and the logical schema (LS) is of a similar nature as the one between LS and CS. As illustrated in Figure 3.2, some performance-oriented physical constructs are *added*

[1] Model-Driven Engineering

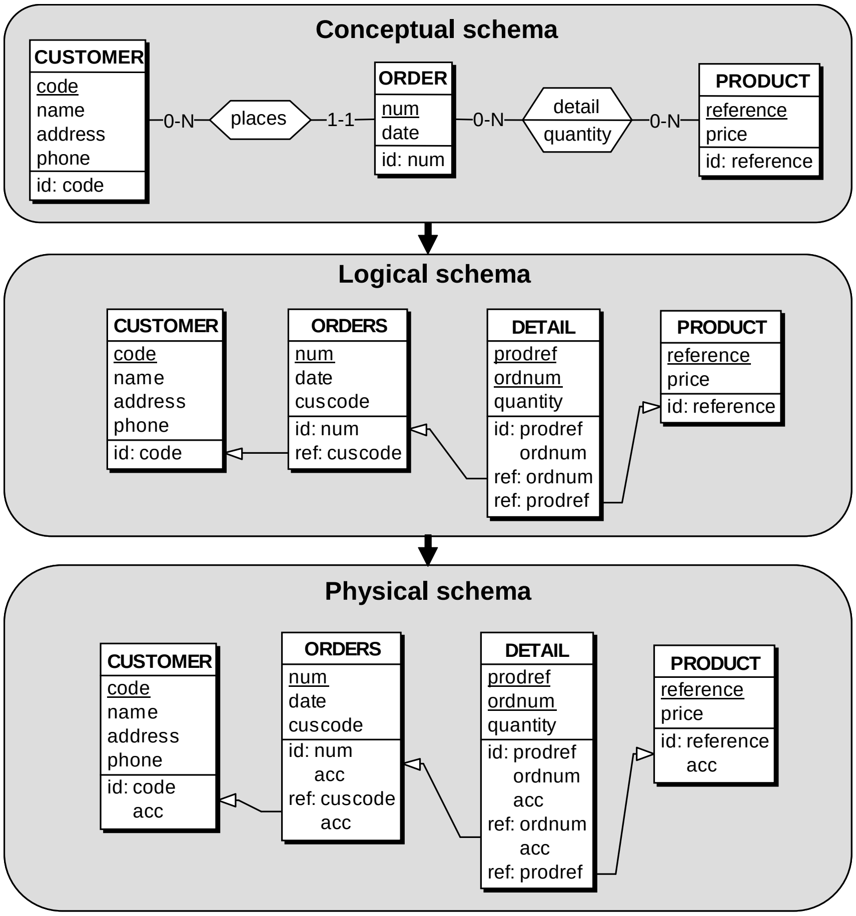


Figure 3.2: Semantically equivalent conceptual, logical and physical schemas.

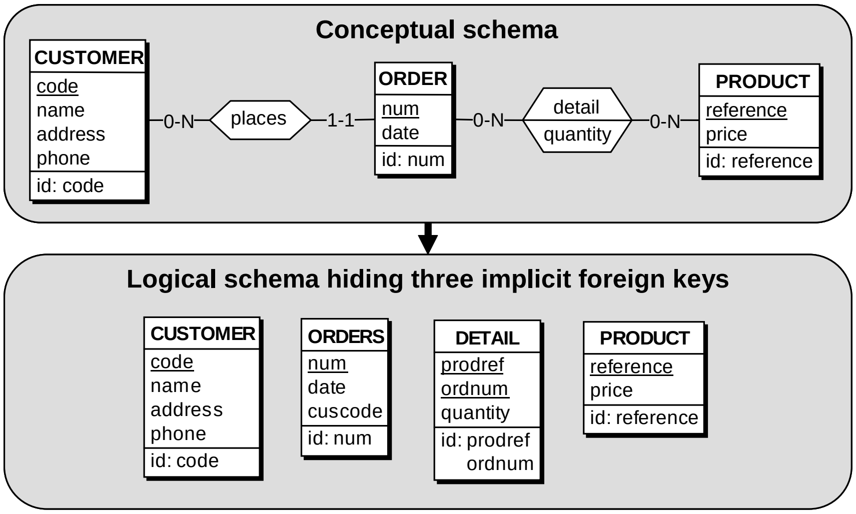


Figure 3.3: Illustration of semantics-decreasing logical design.

during the physical design process (such as indexes in this example). But those additional constructs do not affect the semantics of the schema. Consequently, LS and PS cover the same informational content, i.e., both schemas are semantically equivalent.

DDL code VS physical schema

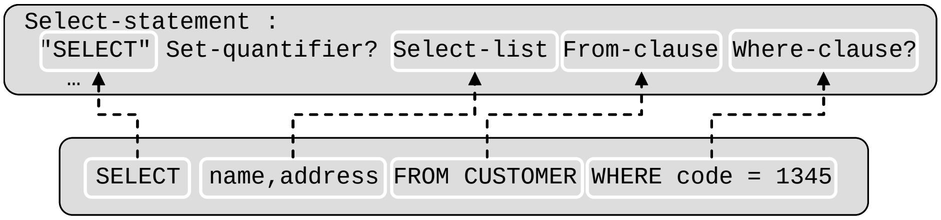
The DDL code is a translation of the physical schema structures and constraints in the target DDL. We say that a DDL code DC is consistent with a physical schema PS if each construct $c \in PS$ corresponds to an equivalent construct $c_{DDL} \in DC$ translating c with high-fidelity. For instance, each SQL table occuring in a relational physical schema should correspond to a `create table...` statement in the DDL code. This consistency relationship is illustrated in Figure 3.4, which shows the standard SQL DDL code corresponding to the physical schema of Figure 3.2.

DDL code VS DDL syntax

Quite naturally, the DDL code should be syntactically correct with respect to the underlying DDL syntax. Although this consistency rule may seem obvious, it constitutes an important practical issue. For instance, the DDL code translating the very same relational schema may significantly differ depending on the chosen DBMS, each defining its own syntactical variations with respect to the SQL standard.

```
create database SCHEMA;

create table CUSTOMER (
   code char(6) not null,
   name char(20) not null,
   address char(40) not null,
   phone numeric(12) not null,
   constraint ID_CUSTOMER_ID primary key (code));

create table DETAIL (
   prodref char(6) not null,
   ordnum char(6) not null,
   quantity numeric(6) not null,
   constraint ID_DETAIL_ID primary key (prodref, ordnum));

create table ORDERS (
   num char(6) not null,
   date date not null,
   cuscode char(6) not null,
   constraint ID_ORDERS_ID primary key (num));

create table PRODUCT (
   reference char(6) not null,
   price numeric(6,2) not null,
   constraint ID_PRODUCT_ID primary key (reference));

alter table DETAIL add constraint REF_DET_ORD_FK
   foreign key (ordnum) references ORDERS;

alter table DETAIL add constraint REF_DET_PRO
   foreign key (prodref) references PRODUCT;

alter table ORDERS add constraint REF_ORD_CUS_FK
   foreign key (cuscode) references CUSTOMER;

create unique index ID_CUSTOMER_IND on CUSTOMER (code);

create unique index ID_DETAIL_IND on DETAIL (prodref, ordnum);

create index REF_DET_ORD_IND on DETAIL (ordnum);

create unique index ID_ORDERS_IND on ORDERS (num);

create index REF_ORD_CUS_IND on ORDERS (cuscode);

create unique index ID_PRODUCT_IND on PRODUCT (reference);
```

Figure 3.4: DDL code corresponding to physical schema of Figure 3.2.



```
Select-statement :
 "SELECT" Set-quantifier? Select-list From-clause Where-clause?
   …

  SELECT  name,address FROM CUSTOMER WHERE code = 1345
```

Figure 3.5: Consistency of a SQL query with respect to the underlying DML syntax.

Data instances VS DDL code

The data instances must comply with the data format and integrity constraints declared in the DDL code. This mainly includes the type, length and number of occurences of column/field values, as well as the uniqueness (identifiers) and referential constraints (foreign keys). Most data management systems offer mechanisms allowing to prevent data inconsistencies to be inserted. Consequently, changing the DDL code typically requires an (often non-trivial) *Extract-Transform-Load* process for propagating DDL code modifications to the data instance level.

Program source code VS DML syntax

The data manipulation statements occuring in the program source code should comply with the data manipulation language syntax. Indeed, database queries are *instances* of corresponding syntax productions, as illustrated in Figure 3.5 for a SQL query. A straighforward way to check such a consistency consists in parsing the DML statements against their expected grammar. Changing the DML grammar or replacing the DML itself necessitates the adaptation of the database queries, which may be non-trivial as we will see in this thesis (in Chapters 7 and 8 in particular).

Program source code VS logical schema

The database queries should also be consistent with the logical schema LS[2]. For instance, each table name occuring in a SQL `from` clause should correspond to a SQL table of the underlying LS. Similarly, each column name occuring in the `select` clause of a query should correspond to a column of at least one of the tables referenced in the `from` clause. This consistency relationship is illustrated in Figure 3.6 in the case of a SQL query.

[2] It may seem more natural to say that queries must comply with the DDL code, which translates the physical schema. However, as described above, the logical schema serves as a sufficient reference for writing valid queries. The additional physical constructs do not influence this task.

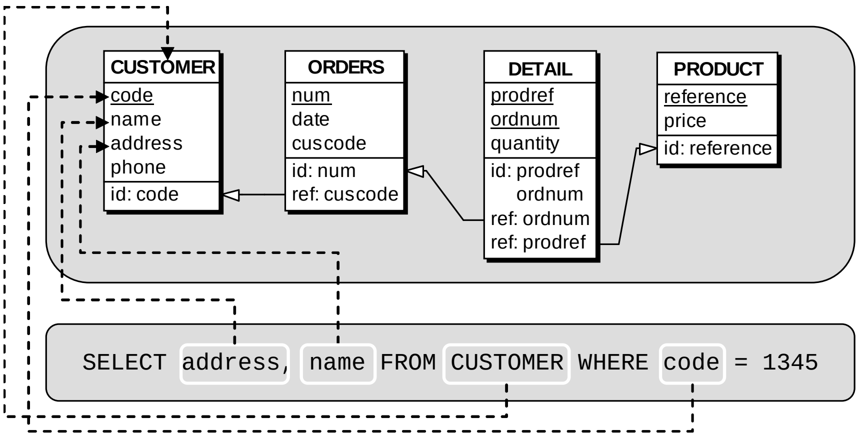


Figure 3.6: Consistency of a SQL query with respect to the underlying logical schema.

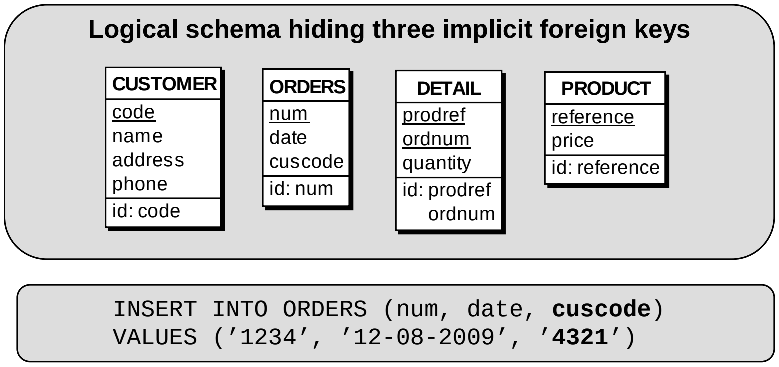
Program execution VS data instances

At compile time, the database queries occuring in the source code of the programs must be consistent with the underlying database schema. At run time, the execution of those queries involves the manipulation of *data instances* of that schema. Each query execution must preserve the consistency of the data with respect to the *implicit schema constructs/constraints* that have to be managed by the application programs, due to the possible semantic loss during logical design.

As an example, Figure 3.7 considers an `insert` query expressed on a logical schema that hides three implicit foreign keys (`ORDERS.cuscode`, `DETAILS.ordnum` and `DETAILS.prodref`). This insert query structurally complies with the SQL syntax, and conforms to the logical schema. However, its execution introduces an inconsistency in the database if the value of `cuscode` ('4321') does not correspond to the `code` of an existing customer. In principle, the program must prevent this problem to arise. Two main data consistency management strategies may be followed:

- *Reactive validation*, according to which verification tasks are systematically performed *before* executing a database modification query that could challenge data consistency. For instance, before inserting a new order, the program verifies that the corresponding customer already exists in the database.

- *Proactive validation*, that consists in enforcing data integrity rules during the query construction process itself, in such a way that the executed database operations never violate integrity constraints. This strategy is typically implemented through user interface restrictions. For example, before placing a new order, the customer must be identified.



```
INSERT INTO ORDERS (num, date, cuscode)
VALUES ('1234', '12-08-2009', '4321')
```

Figure 3.7: Consistency of an `insert` query in the presence of an implicit foreign key.

Summary

Table 3.1 summarizes the above discussion by characterizing the nature of the consistency relationships that must hold between the various artefacts of data-intensive software systems.

3.2 Classification of database evolution scenarios

We propose to classify typical database evolutions according to the three following dimensions:

- *Structural dimension*, that concerns the modification of the database structures (schemas);
- *Semantic dimension*, which relates to the evolution of the semantics of the schemas;
- *Platform/Language dimension*, addressing the replacement of the data description and manipulation languages.

As a first example, Figure 3.8 evaluates the relevance of those three dimensions in the case of three popular evolution scenarios: database *migration*, database *restructuring* and database *integration*.

- *Database migration* consists of the substitution of a data management technology for another one. This scenario raises two major issues. The first one is the conversion of the database schema and instances to a new data management system. The database structure is often modified, but both source and target schema should cover the same *universe of discourse* (i.e., structural modifications but no semantic change). The second problem concerns

Table 3.1: Characterization of the consistency relationships between data-intensive systems artefacts.

	Data model	**DDL syntax**	**DML syntax**	**Conceptual schema**	**Logical schema**	**Physical schema**	**DDL code**	**Data**
Logical schema	Meta-model compliance			Semantic compatibility				
Physical schema					Semantic equivalence			
DDL code		Language consistency				High-fidelity translation		
Data				*Implicit* constraints ompliance			*Explicit* constraints compliance	
Programs			Language consistency		Structural consistency			Semantic consistency

	Structural	**Semantic**	**Platform**
Database migration	✓		✓
Database restructuring	✓	(✓)	
Database integration	✓	✓	(✓)

Figure 3.8: Database evolution scenarios classified according to three dimensions.

the adaptation of the application programs to the migrated database schema and to the target data management system.

- *Database restructuring* does not involve any language replacement, but only structural modifications. Depending on the type of schema transformations applied, the target schema may convey another semantics. For instance, renaming a SQL column does not induce any semantic change while adding a new column does.

- *Database integration* aims at obtaining a single database from heteregeneous databases that belong to the same application domain. The resulting database structure and semantics typically differ from the ones of the input databases. The databases to be integrated are not always of the same platform.

Below, we further present and analyze each of the three identified dimensions of database evolution.

3.2.1 Structural dimension

The structural dimension is concerned with the evolution of the database structures. This regroups modifications applied to the conceptual, logical and physical schemas. As proposed by Hick and Hainaut (2006), we can classify database schema evolutions scenarios according to the schema *initially* modified:

- *Conceptual modifications* (CM) typically translate changes in the functional requirements of the information system into conceptual schema changes.

- *Logical modifications* (LM) do not modify the requirements but adapt their platform-dependent implementation in the logical schema.

- *Physical modifications* (PM) aim at adapting the physical schema to new or evolving technical requirements, like data access performance.

3.2.2 Semantic dimension

The semantic dimension captures the impact of a given database evolution scenario on the informational content of the target database. In other words, it aims at indicating whether the evolution involves:

- *Semantics-augmenting schema modifications* (S^+).

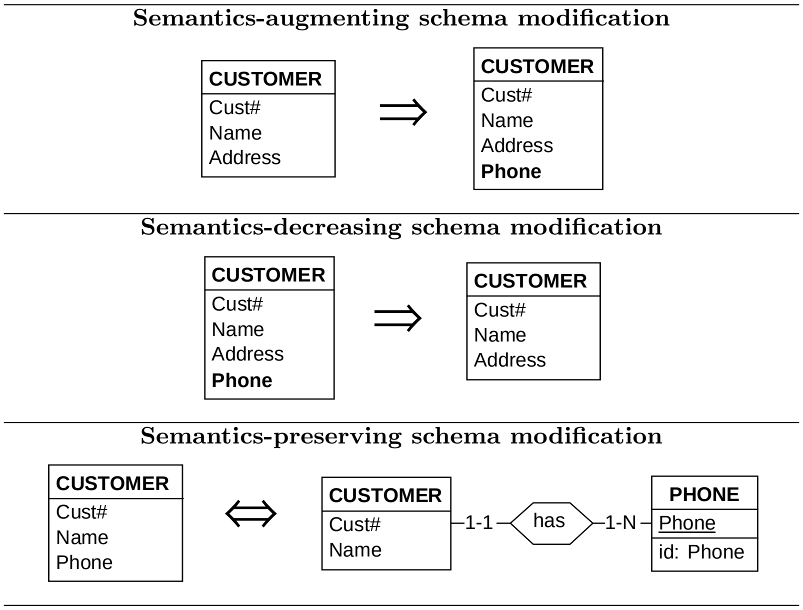


Figure 3.9: Examples of semantics-augmenting, semantics-decreasing and semantics-preserving schema modifications.

- *Semantics-decreasing schema modifications* (S^-);
- *Semantics-preserving schema modifications* ($S^=$);

A semantics-preserving schema modification ($\in S^=$) is commonly called *schema refactoring* (Ambler and Sadalage, 2006). We will regroup the modifications belonging to the two other categories (S^- and S^+) under the term *schema semantic adaptation*. Examples of the three kinds of schema modifications are given in Figure 3.9.

Table 3.2, inspired from (Hick, 2001), provides a semantic classification of the main schema modifications. For the sake of genericity, the presented modifications are based on the GER schema constructs.

3.2.3 Platform/Language dimension

The platform or language dimension intends to characterise a database evolution scenario in terms of platform/language change. We can distinguish three possible cases:

- *Iso-platform evolution* ($P^=$): the database evolution does not involve the replacement of the data management system.
- *Intra-paradigmatic platform change* (P^δ): the database evolution requires the replacement of the data management system with another one belonging to

Schema construct	**Semantic impact of construct modification**		
	S^+	S^-	$S^=$
Entity type	add	remove	rename convert to attribute convert to rel. type split/merge
Relationship type	add	remove	rename convert to ent. type convert to attribute
Role	create increase max. card. decrease min. card. add ent. type	delete decrease max. card. increase min. card. remove ent. type	rename
Is-a relationship	add change type	remove change type	
Attribute	add increase max. card. decrease min. card. extend domain change type	remove decrease max. card. increase min. card. restrict domain change type	rename convert to ent. type aggregate disaggregate instantiate concatenate
Identifier	add add component	remove remove component	rename change type
Constraints	add add component change type	remove remove component change type	rename
Access key	add add attribute	remove remove attribute	rename
Collection	add add ent. type	remove remove ent. type	rename

Table 3.2: Semantic classification of GER schema modifications.

the same paradigm. A typical example is the migration of a MySQL database to Oracle or DB2.

- *Inter-paradigmatic platform change* (P^{Δ}): the database evolution relies on a database paradigm switch, i.e., the migration of the database towards a data management system of another paradigm. This is the case, for instance, when migrating a network database (CODASYL) to a relational platform.

3.2.4 Intra and inter-dimension relationships

The database evolution dimensions we have identified and described so far are obviously related to each other. Based on the direct and indirect consistency relationships depicted in Figure 3.1, we can first identify *intra-dimension* relationships. Indeed, an evolution classified according to one dimension may cause an existing consistency constraint to be violated. The broken consistency link must then be reestablished by means of a *change propagation* evolution, possibly of another kind, which in turn may break another consistency relationship, etc. For instance, within the structural dimension, it usually happens that schema modifications at a given level of abstraction necessitate the adaptation of (1) the schemas belonging to the other abstraction levels, (2) the DDL code, (3) the data instances and (4) the programs.

Furthermore, we can also point out some *inter-dimension* relationships:

- Semantic adaptations typically correspond to conceptual schema modifications.

- Logical modifications are mainly semantics-preserving.

- Inter-paradigmatic platform changes often necessitate logical schema modifications.

3.3 Database evolution processes

Having the above discussion in mind, we will now try to identify the different processes involved in database evolution. This will allow us to better specify the scope and contributions of this thesis. The latter considers that a database evolution is typically composed of the following chain of processes.

1. *Database reverse engineering* is the usual initial step of database evolution, aiming at *understanding* the database subject to evolution. This process is required in the (very) frequent situation in which the database is not (well-)documented.

2. *Database conversion* is the evolution of the database component itself, comprising four subprocesses:

(a) *Impact analysis* aims to evaluate the impact (chain) of the desired schema modification(s) on the related artefacts (other schemas, data instances and programs).

(b) *Schema modification* consists in applying the necessary change(s) to a database schema at a given level of abstraction.

(c) *Schemas adaptation* aims at adapting the schemas belonging to the other abstraction levels[3].

(d) *Data adaptation* concerns the adaptation of the data instances to the modified schemas.

3. *Program adaptation* aims to adapt the application programs to the target database schema and platform.

Note that depending on the database evolution scenario and the chosen methodology, some of the above processes may be useless or ignored.

3.4 Thesis scope, contributions and outline revisited

Focus on database-program consistency

This thesis aims to propose automated techniques for *exploiting* and *preserving* the consistency relationships that hold between a database and the related programs. As previously suggested, the nature of those relationships is threefold:

- *Language consistency*: the database queries of the programs must comply with the underlying data manipulation language;
- *Structural consistency*: the database queries of the programs must comply with the structures and constraints specified in the logical schema;
- *Semantic consistency*: the programs should not introduce data inconsistencies in the database, i.e., data that violate *implicit* schema constructs and constraints.

The role of program analysis and transformation

The automated techniques presented in this thesis rely on *program analysis* and *program transformation.* In the context of the thesis:

- *Program analysis* techniques aim to *exploit semantic consistency* in order to elicitate *implicit* schema constructs and constraints.
- *Program transformation* techniques allow to

[3] For instance, the modification of the conceptual schema typically necessitates the adaptation of the logical schema, physical schemas and DDL code.

Evolution scenario	**Structural dim.**			**Semantic dim.**			**Platform dim.**		
	CM	LM	PM	S^+	S^-	$S^=$	$P^=$	P^{δ}	P^{Δ}
System migration		✓				✓			✓
Schema refactoring		✓				✓	✓		

Figure 3.10: Classification of the database evolution scenarios studied in this thesis.

1. *preserve language consistency* when the data manipulation language changes, e.g., when migrating or upgrading the system towards a new database platform.
2. *preserve structural consistency* when the database schema is refactored, i.e., when semantics-preserving transformations are applied to the schema.

Evolution scenarios considered

The present thesis therefore considers two distinct database evolution scenarios:

- *System migration*, addressed in Chapters 4, 7, 8 and 9 of the thesis.
- *Schema refactoring*, studied in Chapter 10 of the thesis.

Figure 3.10 classifies these two scenarios according to the database evolution dimensions previously identified in this chapter. The considered system migration scenario involves an inter-paradigmatic platform change, which typically necessitates the adaptation of the logical schema through semantics-preserving schema modifications. The schema refactoring scenario does not involve any platform migration, but assumes that semantics-preserving modifications are applied to the logical schema.

Evolution processes considered

The original contributions of the thesis particularly concerns the two following evolution processes:

- *Database reverse engineering*: Chapters 5 and 6 elaborate on the use of program analysis techniques in support to the database reverse engineering process[4].
- *Program adaptation*: Chapters 7, 8 and 10 study the automated propagation of database evolutions to related programs in order to preserve the global consistency of the system.

Roadmap revisited

Figure 3.11 revisits the roadmap of the thesis, by characterizing its chapters in terms of the evolution processes and scenarios they address.

[4] independently of a particular evolution scenario

	Scenario		**Process**			
	System migration	Schema refactoring	Database reverse engineering	Schema modification	Data adaptation	Program adaptation
Chapter 4	✓		✓	✓	✓	✓
Chapter 5			✓			
Chapter 6			✓			
Chapter 7	✓					✓
Chapter 8	✓					✓
Chapter 9	✓		✓	✓	✓	✓
Chapter 10		✓				✓

Figure 3.11: Characterization of the remaining chapters of the thesis in terms of the database evolution processes and scenarios they address.

Part II

The System Migration Problem

Chapter 4

Strategies for Data-Intensive System Migration

Any problem can be solved with a little ingenuity.
– Angus MacGyver

This chapter[1] addresses the problem of platform migration of large business applications, that is, complex software systems built around a database and comprising thousands of programs. More specifically, it studies the substitution of a modern data management technology for a legacy one.

Database platform migration raises two major issues. The first one is the conversion of the database to a new data management paradigm. Recent results have shown that automated lossless database migration can be achieved, both at the schema and data levels (Hick and Hainaut, 2006). The second problem concerns the adaptation of the application programs to the migrated database schema and to the target data management system.

This chapter develops a two-dimensional reference framework that identifies six representative migration strategies. The latter are further analyzed in order to identify methodological requirements. In particular, it appears that transformational techniques are particularly suited to drive the whole migration process. We describe the database migration process, which is a variant of database reengineering. Then, the problem of program conversion is studied. Some migration strategies appear to minimize the program understanding effort, and therefore are sound candidates to develop practical methodologies.

4.1 System migration: State of the Art

Technically, a legacy data-intensive system is made up of large and ageing programs relying on legacy database systems (like IMS or CODASYL) or using primitive

[1] An extended version of this chapter appeared in the book *Software Evolution*, published by Springer in January 2008 (Hainaut et al., 2008).

DMSs[2] (a.o., COBOL file system, ISAM). Such systems often are isolated in that they do not easily interface with other applications. Moreover, they have proved critical to the business of organizations. To keep being competitive, organizations must improve their legacy systems and invest in advanced technologies, specially through system evolution. In this context, the claimed 75% cost of legacy systems maintenance (w.r.t. total cost) is considered prohibitive (Wiederhold, 1995).

System migration is an expensive and complex process, but it greatly increases the system control and evolution to meet future business requirements. The scientific and technical literature (e.g. Bisbal et al., 1999; Brodie and Stonebraker, 1995) mainly identifies two migration strategies, namely rewriting the legacy system from scratch or migrating by small incremental steps. The incremental strategy allows the migration projects to be more controllable and predictable in terms of calendar and budget. The difficulty lies in the determination of the migration steps.

Legacy system migration is a major research domain that has yielded some general migration methods. For example, Tilley and Smith (1995) discuss current issues and trends in legacy system reengineering from several perspectives (engineering, system, software, managerial, evolutionary, and maintenance). They propose a framework to place reengineering in the context of evolutionary systems. The butterfly methodology proposed by Wu et al. (1997) provides a migration methodology and a generic toolkit to aid engineers in the process of migrating legacy systems. This methodology, that does not rely on an incremental strategy, eliminates the need of interoperability between the legacy and target systems.

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

4.1.1 Language dimension

Language conversion consists in translating (parts of) an existing program from a source programming language to a target programming language. Ideally, the target program should show the same behaviour as the source program. Malton (2001) identifies three kinds of language conversion scenarios, with their own difficulties and risks:

- **Dialect conversion** is the conversion of a program written in one dialect of a programming language to another dialect of the same programming language.

- **API migration** is the adaptation of a program due to the replacement of external APIs. This problem may prove challenging, even when the source and target APIs are similar to each other, as shown by Lämmel and van der Storm (2009).

- **Language migration** is the conversion from one programming language to a different one. It may include dialect conversion and API migration.

[2] DMS: Data Management System.

Two main language conversion approaches can be found in the literature. The first one (Waters, 1988), that might be called *abstraction-reimplementation*, is a two-step method. First, the source program is analyzed in order to produce a high-level, language-independent description. Second, the reimplementation process transforms the abstract description obtained in the first step into a program in the target language. The second conversion approach (Terekhov and Verhoef, 2000; Malton, 2001) does not include any abstraction step. It is a three-phase conversion process: (1) *normalization*, that prepares the source program to make the translation step easier; (2) *translation*, that produces an equivalent program that correctly runs in the target language; (3) *optimization*: that improves the maintenability of the target source code

Terekhov and Verhoef (2000) show that the language conversion process is far from trivial, especially when the source and the target languages belong to different paradigms. A lot of research has been carried out on specific cases of language conversion, among which PL/I to C++ (Kontogiannis et al., 1998), Smalltalk to C (Yasumatsu and Doi, 1995), C to Java (Martin and Müller, 2001) and Java to C# (El-Ramly et al., 2006).

4.1.2 User interface dimension

Migrating user interfaces to modern platforms is another popular migration scenario. Such a process may often benefit from an initial reverse engineering phase, as shown in the method proposed by Stroulia et al. (2003). This method starts from a recorded trace of the user interaction with the legacy interface, and produces a corresponding state-transition model. The states represent the unique legacy interface screens while the transitions correspond to the user action sequences enabling transitions from one screen to another. Lucia et al. (2006) propose a practical approach to migrating legacy systems to multi-tier, web-based architectures. They present an Eclipse-based plugin to support the migration of the graphical user interface and the restructuring and wrapping of the original legacy code.

4.1.3 Platform and architecture dimensions

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

Towards distributed architectures The Renaissance project (Warren, 1999) develops a systematic method for system evolution and re-engineering and provides technical guidelines for the migration of legacy systems (e.g. COBOL) to distributed client/server architectures. A generic approach to reengineering legacy code for distributed environments is presented by Serrano et al. (2002). The methodology combines techniques such as data mining, metrics, clustering, object identification and wrapping. Canfora et al. (2006) propose a framework supporting the development of thin-client applications for limited mobile devices. This framework allows Java AWT applications to be executed on a server while the graphical interfaces are displayed on a remote client.

Towards object-oriented platforms Migrating legacy systems towards object-oriented structures is another research domain that has led to a lot of mature results, especially on object identification approaches (Yeh et al., 1995; Canfora et al., 1996; van Deursen and Kuipers, 1999; Girard et al., 1999; Sahraoui et al., 1999). Regarding the migration process itself, the approach suggested by de Lucia et al. (1997) consists of several steps combining reverse engineering and reengineering techniques. More recently, Zou and Kontogiannis (2001) have presented an incremental and iterative migration framework for reengineering legacy procedural source code into an object-oriented system.

Towards aspect-orientation A significant research effort was recently devoted to the migration of legacy systems towards aspect-oriented programming (AOP). Several authors have addressed the initial reverse engineering phase of the process, called *aspect mining*, which aims at identifying crosscutting concern code in existing systems. Among the various aspect mining techniques that have been proposed, we mention *fan-in analysis* (Marin et al., 2004), *formal concept analysis* (Tourwe and Mens, 2004), *dynamic analysis* (Tonella and Ceccato, 2004) and *clone detection* (Bruntink et al., 2005). While those techniques still suffer from some limitations in terms of precision and recall (Mens et al., 2008), Ceccato et al. (2006) showed that combining them may allow to reach a more complete coverage of concerns.

The second step of the migration process consists in refactoring the detected cross-cutting concern code into aspects. While such a refactoring is feasible Binkley et al. (2006), it is important to evaluate its benefits for a particular system. As observed by Bruntink et al. (2007), the level of source code variability for *"simple" cross-cutting concerns* - like tracing, parameter checking or exception handling - may be very high in legacy systems. This situation tends to reduce the benefits of migrating the cross-cutting concern code to aspects and is also makes such a process more risky, especially when a strict equivalence of the source and target systems is imposed.

Towards service-oriented architectures Migrating legacy systems towards service-oriented architectures (SOA) appears as one of the next challenges of the maintenance community. Sneed (2006) presents a wrapping-based approach according to which legacy program functions are offered as web services to external users. Liam O'Brien (2005) propose the use of architecture reconstruction to support migration to SOA. Heckel et al. (2008) present a tool-supported methodology for migrating legacy systems towards three-tier and service-oriented architectures. This approach is based on graph transformation technology.

4.1.4 Database dimension

Closer to our data-centered approach, the Varlet project (Jahnke and Wadsack, 1999) adopts a typical two phase reengineering process comprising a reverse engineering phase followed by a standard database implementation. The approach of Jeusfeld and Johnen (1994) is divided into three parts: mapping of the original schema into a meta model, rearrangement of the intermediate representation and production of the target schema. Some works also address the migration between two specific systems. Among those, Menhoudj and Ou-Halima (1996) present a method to migrate the data of COBOL legacy system into a relational database management system. The hierarchical to relational database migration is discussed by Meier et al. (1994); Meier (1995). General approaches to migrate relational database to object-oriented technology are proposed by Behm et al. (1997) and Missaoui et al. (1998). More recently, Bianchi et al. (2000) propose an iterative approach to database reengineering. This approach aims at eliminating the *ageing symptoms* of the legacy database (Visaggio, 2001) when incrementally migrating the latter towards a modern platform.

4.1.5 Discussion

Though the current literature on data-intensive systems migration sometimes recommends a semantics-based approach, relying on reverse engineering techniques, most technical solutions adopted in the industry are based on the so-called *one-to-one* migration of the data structures and contents, through a fully-automated process. As we will see in the remaining of this chapter, these approaches lead to poor quality results. Secondly, while most papers provide ad hoc solutions for particular combinations of source/target platforms, there is still a lack of generic and systematic studies encompassing database migration strategies and techniques. Thirdly, the conversion of application programs in the context of database migration still remains an open problem. Although some work (e.g. Bianchi et al., 2000) suggests the use of wrapping techniques, very little attention is devoted to the way database wrappers are built or, better, automatically generated. In addition, the impact of the chosen program conversion technique on target source code maintainability has not been sufficiently discussed.

4.2 Migration reference model

There is more than one way to migrate a data-intensive software system. Some approaches are quite straightforward and inexpensive, but lead to poorly structured results that are difficult to maintain. Others, on the contrary, produce good quality data structures and code, but at the expense of substantial intelligent (and therefore difficult to automate) code restructuring. We have built a reference model based on two dimensions, namely data and programs. Each of them defines a series of change strategies, ranging from the simplest to the most sophisticated. This model outlines a solution space in which we identify six typical strategies that will be

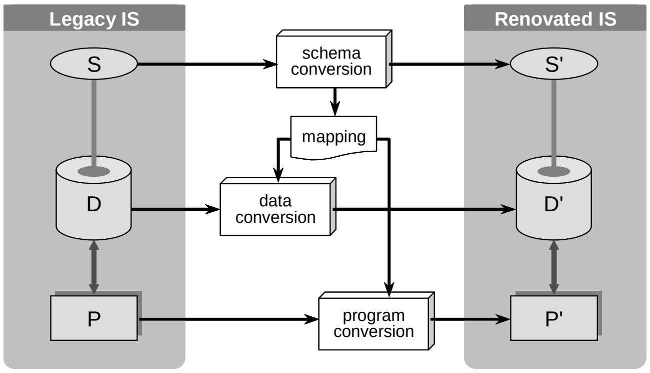


Figure 4.1: Overall view of the *database-first* system migration process

described below and discussed in the remainder of the chapter. This model relies on a frequently used scenario, called *database-first* (Wu et al., 1997), according to which the database is transformed before program conversion. This approach allows developers to cleanly build new applications on the new database while incrementally migrating the legacy programs.

In this context, system migration consists in deriving a new database from a legacy database and in further adapting the software components accordingly (Brodie and Stonebraker, 1995). Considering that a database is made up of two main components, namely its schema(s) and its contents (the *data*), the migration comprises three main steps: (1) *schema conversion*, (2) *data conversion* and (3) *program conversion*. Figure 4.1 depicts the organization of the database-first migration process, that is made up of subprocesses that implement these three steps. Schema conversion produces a formal description of the mapping between the objects of the legacy (S) and renovated (S') schemas. This mapping is then used to convert the data and the programs. Practical methodologies differ in the extent to which these processes are automated.

- **Schema conversion** is the translation of the legacy database structure, or schema, into an equivalent database structure expressed in the new technology. Both schemas must convey the same semantics, i.e., all the source data should be losslessly stored into the target database. Most generally, the conversion of a source schema into a target schema is made up of two processes. The first one, called database reverse engineering (Hainaut et al., 1996a), aims at recovering the conceptual schema that expresses the semantics of the source data structure. The second process is standard and consists in deriving the target physical schema from this conceptual specification. Each of these processes can be modeled by a chain of semantics-preserving schema transformations.

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

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

4.2.1 Strategies

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

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

The Program dimension (P) Once the database has been converted, several approaches to application programs adaptation can be followed. We identify three reference strategies. The first one (*Wrappers* or P1) relies on wrappers that encapsulate the new database to provide the application programs with the legacy data access logic, so that these programs keep reading and writing records in (now fictive) indexed files or CODASYL/IMS databases, generally through program calls instead of through native I/O file statements. The second strategy (*Statement rewriting* or P2) consists in rewriting the access statements in order to make them process the new data through the new DMS-DML [3]. For instance, a COBOL `READ` statement is replaced with a select-from-where (SFW) or a fetch SQL statement. In these two first strategies, the program logic is neither elicited nor changed. According to the third strategy (*Logic rewriting* or P3), the program is rewritten in order to use the new DMS-DML at its full power. It requires a deep understanding of the program logic, since the latter will generally be changed due to, for instance,

[3] DML: Data Manipulation Language.

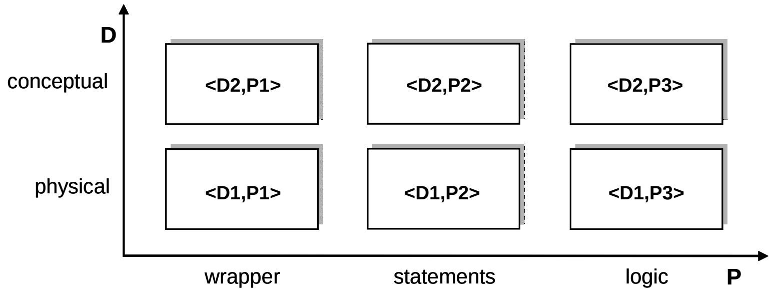


Figure 4.2: The six reference IS migration strategies

the change in database paradigm. These dimensions define six reference system migration strategies (Figure 4.2).

4.2.2 Running example

The strategies developed in this chapter will be illustrated by a small case study in which the legacy system comprises a standalone COBOL program and three files. Despite its small size, the files and the program exhibit representative instances of the most problematic patterns. This program records and displays information about customers that place orders. The objective of the case study is to convert the legacy files into a new relational database and to transform the application program into a new COBOL program, with the same business functions, but that accesses the new database.

4.3 Schema conversion

The schema conversion strategies mainly differ in the way they cope with the explicit and implicit constructs (that is, the data structures and the integrity constraints) of the source schema. An *explicit construct* is declared in the DDL code [4] of the schema and can be identified through examination or parsing of this code. An *implicit construct* has not been declared, but, rather, is controlled and managed by external means, such as decoding and validating code fragments scattered throughout the application code. Such a construct can only be identified by sophisticated analysis methods exploring the application code, the data, the user interfaces, to mention the most important sources.

The schema conversion process analyzes the legacy application to extract the source physical schema (SPS) of the underlying database and transforms it into a target physical schema (TPS) for the target DMS. The TPS is used to generate the DDL code of the new database. In this section, we distinguish two schema conversion strategies. The first strategy, called the *physical* schema conversion,

[4] DDL: Data Description Language.

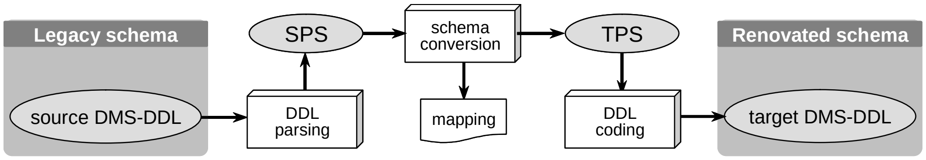


Figure 4.3: Physical schema conversion strategy (D1).

merely simulates the explicit constructs of the legacy database into the target DMS. According to the second one, the *conceptual* schema conversion, the complete semantics of the legacy database is retrieved and represented into the technology-neutral conceptual schema (CS), which is then used to develop the new database.

4.3.1 Physical conversion strategy (D1)

4.3.1.1 Principle

According to this strategy (Figure 4.3) each explicit construct of the legacy database is directly translated into its closest equivalent in the target DMS. For instance, considering a standard file to SQL conversion, each record type is translated into a table, each top-level field becomes a column and each record/alternate key is translated into a primary/secondary key. No conceptual schema is built, so that the semantics of the data is ignored.

4.3.1.2 Methodology

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

4.3.1.3 Illustration

The analysis of the file and record declarations produces the SPS (Figure 4.4/left). Each COBOL record type is translated into an SQL table, each field is converted into a column and object names are made compliant with the SQL syntax (Figure 4.4/right). In this schema, a box represents a physical entity type (record type, table, segment, etc.). The first compartment specifies its name, the second one gives its components (fields, columns, attributes) and the third one declares secondary constructs such as keys and constraints (*id* stands for primary identifier/key, *acc* stands for access key, or index, and ref stands for foreign key). A cylinder represents a data collection, commonly called a file.

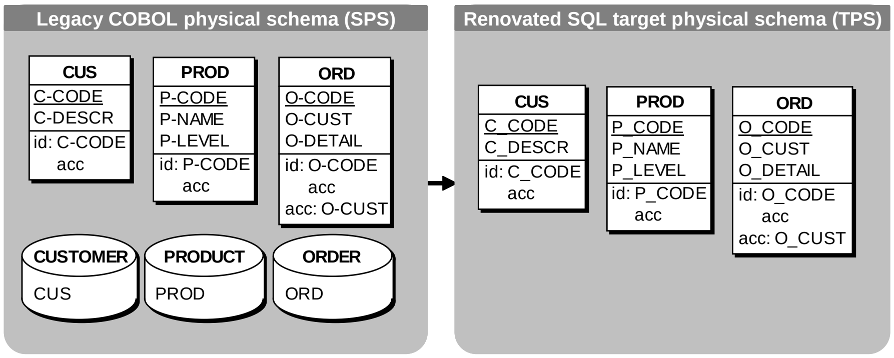


Figure 4.4: Example of COBOL/SQL physical schema conversion.

4.3.2 Conceptual conversion strategy (D2)

4.3.2.1 Principle

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

4.3.2.2 Methodology

The left part of Figure 4.5 depicts the three steps of a simplified database reverse engineering methodology used to recover the logical and conceptual schemas of the source database.

- As in the first strategy, the first step is the parsing of the DDL code to extract the physical schema (SPS), which only includes the explicit constructs.

- The *schema refinement* step consists in refining the SPS by adding the implicit constructs that are identified through the analysis of additional information sources, such as the source code of the application programs and the database contents, to mention the most common ones. Program code analysis performs an in-depth inspection of the way the programs use and manage the data. Data validation, data modification and data access programming *clichés* are searched for in particular, since they concentrate the procedural logic strongly linked with data properties. The existing data are

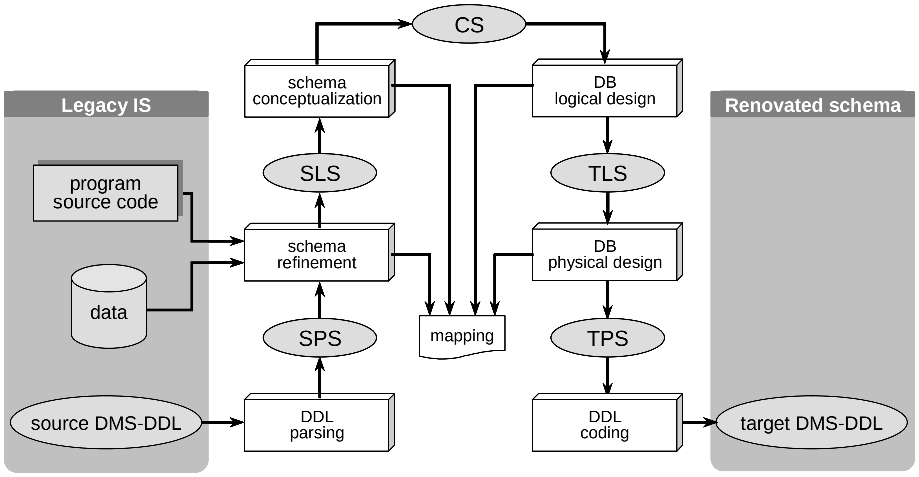


Figure 4.5: Conceptual schema conversion strategy (D2)

also analyzed through data mining techniques, either to detect constraints, or to confirm or discard hypotheses on the existence of constraints. This step results in the source logical schema (SLS), that includes the explicit representation of such constructs as record and field decomposition, uniqueness constraints, foreign keys or enumerated domains that were absent in SPS. The history SPS-to-SLS of the refinement process forms the first part of the source-to-target mapping.

- The final step is *schema conceptualization* that semantically interprets the logical schema. The result is expressed by the conceptual schema (CS). This schema is technology independent, and therefore independent of both the legacy and new DMSs. The history SLS-to-CS of this process is appended to the source-to-target mapping.

A complete presentation of this reverse engineering methodology can be found in Hainaut et al. (1996a) and Hainaut (2002), together with a fairly comprehensive bibliography on database reverse engineering.

The conceptual schema is then transformed into an equivalent logical schema (TLS), which in turn is transformed into the physical schema (TPS). TPS is then used to generate the DDL code of the target database. These processes are quite standard and are represented in the right part of Figure 4.5. The histories CS-to-TLS and TLS-to-TPS are added to the source-to-target mapping. The mapping SPS-to-TPS is now complete, and is defined as SPS-to-SLS ∘ SLS-to-CS ∘ CS-to-TLS ∘ TLS-to-TPS.

4.3.2.3 Illustration

The details of this reverse engineering case study have been described in Hainaut et al. (1997). We sketch its main steps in the following. The legacy physical schema SPS is extracted as in the first approach (Figure 4.6/top-left).

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

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

(2) The implicit foreign keys `O-CUST` and `REF-DET-PRO` are identified by schema names and structure patterns analysis, program code analysis and data analysis.

(3) The multivalued identifier (uniqueness constraint) `REF-DET-PRO` of `O-DETAIL` can be recovered through the same techniques.

The resulting logical schema SLS is depicted in Figure 4.6/top-right.

During the *data structure conceptualization*, the implementation objects (record types, fields, foreign keys, arrays,...) are transformed into their conceptual equivalent to produce the conceptual schema CS (Figure 4.6/bottom-left).

Then, the database design process transforms the entity types, the attributes and the relationship types into relational constructs such as tables, columns, keys and constraints. Finally physical constructs (indexes and storage spaces) are defined (Figure 4.6.bottom-right) and the code of the new database is generated.

4.4 Data conversion

4.4.1 Principle

Data conversion is handled by a so-called Extract-Transform-Load (ETL) processor (Figure 4.7), which transforms the data from the data source to the format defined by the target schema. Data conversion requires three steps. First, it performs the extraction of the data from the legacy database. Then, it transforms these data in such a way that their structures match the target format. Finally, it writes these data in the target database.

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

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

The conceptual schema conversion strategy (D2) recovers the conceptual schema (CS) and the target physical schema (TPS) implements all the constraints of this schema. Generally, both CS and TPS include constraints that are missing in SPS,

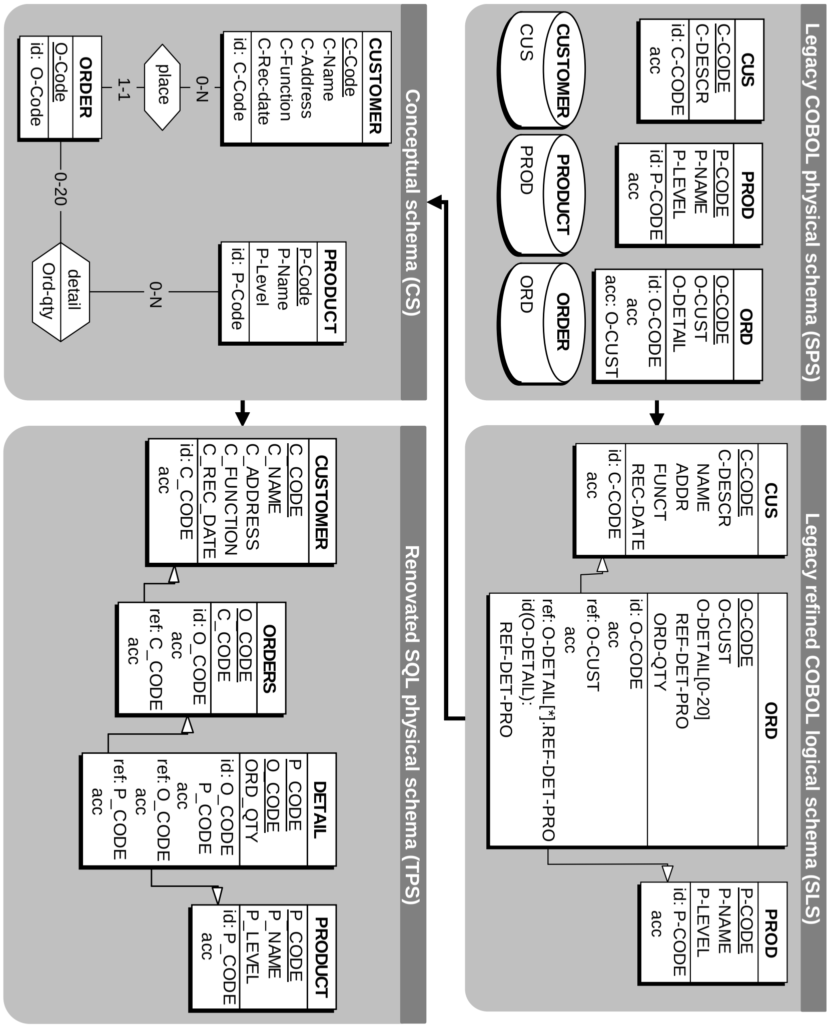


Figure 4.6: Example of COBOL/SQL conceptual schema conversion.

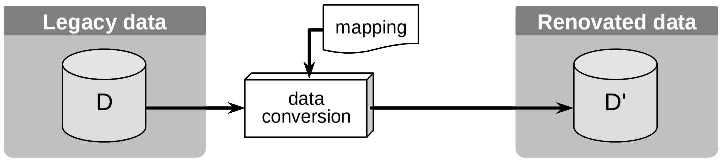


Figure 4.7: Mapping-based data migration architecture.

and that the source data may violate. Thus data migration must include a preliminary data cleaning step that fixes or discards the data that cannot be loaded in the target database (Rahm and Do, 2000). This step cannot always be automated. However, the schema refinement step identifies all the implicit constraints and produces a formal specification for the data cleaning process. It must be noted that the physical schema conversion strategy (D1) makes such data cleaning useless. Indeed, both SPS and TPS express the same constraints that the source data are guaranteed to satisfy.

4.4.2 Methodology

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

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

4.5 Program conversion

The program conversion process aims at re-establishing the consistency relationships that hold between application programs and the migrated database. As described in Chapter 3, the nature of this consistency is twofold. First, the programs have to comply with the API of the DMS, by using the right data manipulation language and interaction protocols. Second, the programs have to manipulate the data in their correct format, i.e., the format declared in the database schema.

This section analyzes the three program modification strategies specified in Figure 4.2. The first one relies on *wrapper technology* (P1) to map the access primitives onto the new database through wrapper invocations that replace the DML statements of the legacy DMS. The second strategy (P2) replaces each statement with its equivalent in the new DMS-DML. According to the P3 strategy, the access logic is rewritten to comply with the DML of the new DMS. In strategies P2 and P3, access statements are expressed in the DML of the new DMS.

In order to compare the three program conversion strategies, we will apply them successively on the same legacy COBOL fragment, given in Figure 4.8. This code fragment deletes all the orders placed by a given customer.

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

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

Figure 4.8: A legacy COBOL code fragment that deletes the orders corresponding to a given customer.

4.5.1 Wrapper strategy (P1)

4.5.1.1 Principle

In migration and interoperability architectures, wrappers are popular components that convert legacy interfaces into modern ones. Such wrappers allow the reuse of legacy components (Sneed, 2000) (e.g., allow Java programs to access COBOL files). The wrappers discussed in this chapter are of a different nature, in that they simulate the legacy data interface on top of the new database. For instance, they allow COBOL programs to *read*, *write*, *rewrite* records that are built from rows extracted from a relational database. In a certain sense, they could be called *backward* wrappers. An in-depth analysis of both kinds of wrappers can be found in Thiran et al. (2006).

The wrapper conversion strategy attempts to preserve the logic of the legacy programs and to map it on the new DMS technology (Brodie and Stonebraker, 1995). A *data wrapper* is a *data model conversion* component that is called by the application program to carry out operations on the database. In this way, the application programs invoke the wrapper instead of the legacy DMS. If the wrapper simulates the modeling paradigm of the legacy DMS and its interface, the alteration of the legacy code is minimal and largely automatable. It mainly consists in replacing DML statements with wrapper invocations.

The wrapper converts all legacy DMS requests from legacy applications into requests against the new DMS that now manages the data. Conversely, it captures results from the new DMS, converts them to the appropriate legacy format (Papakonstantinou et al., 1995) (Figure 4.9) and delivers them to the application program.

4.5.1.2 Methodology

Schemas SPS and TPS, as well as the mapping between them (SPS-to-TPS) provide the necessary information to derive the procedural code of the wrappers. For each

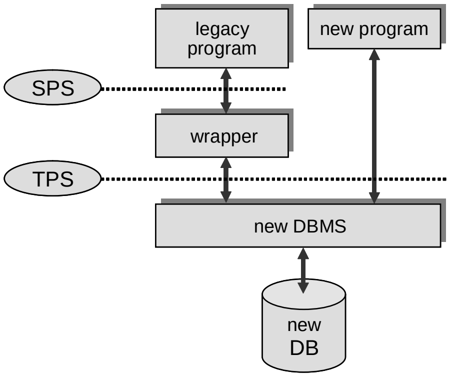


Figure 4.9: Wrapper-based migration architecture: a wrapper allows the data managed by a new DMS to be accessed by the legacy programs.

COBOL source record type, a wrapper is built that simulates the COBOL file handling statements. The simulated behaviour must also include the management of currency indicators (internal dynamic pointers to current records) as well as error handling.

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

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

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

The <D1,P1> and <D2,P1> strategies only differ in the complexity of the wrappers that have to be generated. The program transformation is the same in both strategies since each legacy DML instruction is replaced with a wrapper invocation. The code of the wrappers for the <D1,P1> strategy is trivial because each explicit data structure of the legacy database is directly translated into a similar structure of the target database. In the <D2,P1> strategy the conceptual schema is recovered and the new physical schema can be very different from the

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

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

Figure 4.10: Code fragment of Fig. 4.8 converted using the *Wrapper* strategy (P1)

legacy one. For instance, a record can be split into two or more tables, a table may contain data from more than one record, new constraints might be implemented into the new DMS, etc. In this strategy, translating a READ command may require to access more than one table and to perform additional tests and loops.

4.5.1.3 Illustration

To illustrate the way data wrappers are used, let us consider the legacy COBOL fragment of Figure 4.8, which comprises READ and DELETE primitives. As shown in Figure 4.10, each primitive is simply replaced with a corresponding wrapper invocation. From the program side, the wrapper forms a black box that simulates the behaviour of the COBOL file handling primitives on top of the SQL database. Note that the P1 program adaptation strategy does not depend on the schema conversion strategy. This choice only affects the complexity of the wrapper code, since the latter is directly derived from the mapping that holds between the legacy and new database schemas.

4.5.2 Statement rewriting strategy (P2)

4.5.2.1 Principle

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

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

4.5.2.2 Methodology

The program modification process may be technically complex, but does not need sophisticated methodology. Each DML statement has to be located, its parameters have to be identified and the new sequence of DML statements has to be defined and inserted in the code. The main point is how to translate iterative accesses in a systematic way. For instance, in the most popular COBOL-to-SQL conversion, there exist several techniques to express the typical `START/READ NEXT` loop with SQL statements. The task may be complex due to loosely structured programs and the use of dynamic DML statements. For instance, a COBOL `READ NEXT` statement can follow a statically unidentified `START` or `READ KEY IS` initial statement, making it impossible to identify the record key used. A description of a specific technique that solves this problem is provided below.

4.5.2.3 Illustration

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

The technique illustrated in Figure 4.11 solves this problem. This technique is based on state registers, such as `ORD-SEQ`, that specify the current key of each record type, and consequently the matching SQL cursor. A cursor is declared for each kind of record key usage (*equal*, *greater*, *not less*) in the program. For instance, the table `ORD` gives at most six cursors (combination of two record keys and three key usages).

The example of Figure 4.11 shows the <D2,P2> conversion the COBOL code fragment of Figure 4.8. During the schema conversion process, the `O-DETAIL` compound multivalued field has been converted into the `DETAIL` SQL table. So, rebuilding the value of `O-DETAIL` requires the execution of a loop and a new `FILL-ORD-DETAIL` procedure. This new loop retrieves the details corresponding to the current `ORD` record, using a dedicated SQL cursor.

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

Figure 4.11: Code fragment of Fig. 4.8 converted using the *Statement Rewriting* strategy (P2)

4.5.3 Logic rewriting strategy (P3)

4.5.3.1 Principle

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

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

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

4.5.3.2 Methodology

The P3 strategy is much more complex than the previous ones since every part of the program may be influenced by the schema transformation. The most obvious method consists in (1) identifying the file access statements, (2) identifying and understanding the statements and the data objects that depend on these access statements and (3) rewriting these statements as a whole and redefining these data objects.

4.5.3.3 Illustration

Figure 4.12 shows the code fragment of Figure 4.8 converted using the *Logic Rewriting* strategy. The resulting code benefits from the full power of SQL. The two-step *position then delete* pattern, which is typical of navigational DMS, can be replaced with a single predicate-based *delete* statement.

4.6 Strategies comparison

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

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

Figure 4.12: Code fragment of Fig. 4.8 converted using the *Logic Rewriting* strategy (P3)

4.6.1 Database conversion strategies

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

Nevertheless, we must mention an unfrequent situation for which this strategy can be valuable, that is, when the legacy database has been designed and implemented in a disciplined way according to the database theory. For instance, a database made up of a collection of 3NF [5] record types can be migrated in a straighforward way to an equivalent relational database of good quality.

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

[5] 3NF stands for *third normal form*

4.6.2 Program conversion strategies

The *wrapper strategy* (P1) does not alter the logic of the legacy application program. When working on the external data, the transformed program simply invokes the wrapper instead of the legacy DMS primitives. The transformation of the program is quite straightforward: each legacy DMS-DML is replaced with a call to the wrapper. So, this transformation can easily be automated. The resulting program has almost the same code as the source program, so a programmer who has mastered the latter can still maintain the new version without any additional effort or documentation. When the structure of the database evolves, only the wrapper need be modified, while the application program can be left unchanged. The complexity of the wrapper depends on the strategy used to migrate the database. In the D1 strategy, the wrapper is quite simple: it reads one line of the table, converts the column values and produces a record. In the D2 strategy, the wrapper can be very complex, since reading one record may require complex joins and loops to retrieve all the data. Despite the potentially complex mapping between SPS and TPS, which is completely encapsulated into the wrapper, the latter can be produced automatically, as shown by Thiran et al. (2006). A wrapper may induce computing and I/O overhead compared to P2 and P3 strategies.

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

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

4.6.3 System migration strategies

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

- **<D1,P1>**: This approach produces a (generally) badly structured database that will suffer from poor performance but preserves the program logic, notably because the database interface is encapsulated in the wrapper. It can be recommended when the migration must be completed in a very short time, e.g., when the legacy environment is no longer available. Developing new applications should be delayed until the correct database is available. This approach can be a nice first step to a better architecture such as that produced by <D2,P1>. However, if the legacy database already is in 3NF, the result is close to that of strategy <D2,P1>.

- **<D2,P1>**: This strategy produces a good quality database while preserving the program logic. New quality applications can be developed on this database. The legacy programs can be renovated later on, step by step. Depending on the impedance mismatch between the legacy and target technologies, performance penalty can be experienced. For instance, wrappers that simulate CODASYL DML on top of a relational database have to synchronize two different data manipulation paradigms, a process that may lead to significant data access overhead.

- **<D1,P2>**: Despite its popularity, due to its low cost, this approach clearly is the worst one. It produces a database structure that is more obscure than the source one, and that provides poorer performance. The programs are inflated with obscure data management code that makes them complex and more difficult to read, understand and maintain. Such a renovated system cannot evolve at sustainable cost, and therefore has no future. If the legacy database already is in 3NF, the result may be similar to that of strategy <D2,P2>.

- **<D2,P2>**: Produces a good quality database, but the programs can be unreadable and difficult to maintain. It can be considered if no maintenance of the application is planned and the programs are to be rewritten in the near future. If the wrapper overhead is acceptable, the<D2,P1> strategy should be preferred.

- **<D1,P3>**: Data migration produces a very poor quality database that simulates the legacy database. Adapting, at high cost, the program to these awkward structures is meaningless, so that we can consider this strategy not pertinent

- **<D2,P3>**: This strategy provides both a database and a set of renovated programs of high quality, at least as far as database logic is concerned. Its cost also is the highest. This is a good solution if the legacy program language is kept and if the programs have a clean and clear structure.

4.7 Conclusions

The variety in corporate requirements, as far as system reengineering is concerned, naturally leads to a wide spectrum of migration strategies. This chapter has identified two main independent lines of decision, the first one related to the precision of database conversion (schema and contents) and the second one related to program conversion. From them, we were able to identify and analyze six reference system migration strategies. The thorough development of these technical aspects is the major contribution of this chapter since most of these aspects have only been sketched in the literature (Brodie and Stonebraker, 1995).

Despite the fact that a supporting technology has been developed[6], and therefore makes some sophisticated strategies realistic at an industrial level, we still lack sufficient experience to suggest application rules according to the global corporate strategy and to intrinsic properties of the legacy system. As is now widely accepted in maintenance, specific metrics must be identified to score the system against typical reference patterns. Such criteria as the complexity of the database schema, the proportion of implicit constructs, the underlying technology, or the normalisation level, to mention only a few, should certainly affect the feasibility of each migration strategy. Corporate requirements like performance, early availability of components of the renovated system, program independence against the database structure, skills of the development team, or availability of human resources are all aspects that could make some strategies more valuable than others.

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

Roadmap

In this chapter, we have seen that the D2 database conversion strategy allows to produce a high-quality target database, thanks to an initial reverse engineering process. In the next part of this thesis (Part III), we will explore the use of program analysis techniques for supporting this initial phase. Then, in Part IV, we will present DMS-specific methods and tools for supporting the program conversion phase of database platform migration, and discuss the application of those tool-supported methods in the context of industrial migration projects.

[6] as we will see in Part IV.

Part III

Program Analysis for Database Reverse Engineering

Chapter 5

Static Dependency Analysis

Little by little, one travels far.
– J. R. R. Tolkien

This chapter[1] focuses on the use of static program analysis in the context of database reverse engineering. It presents a general, tool-supported dataflow analysis methodology dedicated to data-intensive software systems. It shows how the approach and tools allow to contribute to the recovery of implicit schema constructs, by discussing their application in the context of industrial reverse engineering projects.

5.1 Introduction

Data(base) Reverse Engineering (DBRE) is "*a collection of methods and tools to help an organization determine the structure, function, and meaning of its data*"(Chikofsky, 1996). In particular, DBRE aims at recovering the precise semantics of the data, by retrieving *implicit* data constructs and constraints, i.e., not explicitly declared in the database declaration but verified in the procedural code (Hainaut et al., 2000), among which:

- fine-grained structure of entity types and attributes;
- referential constraints (foreign keys);
- exact cardinalities of attributes;
- identifiers of multi-valued attributes.

Database reverse engineering is a complex and expensive task, that needs to be supported by program understanding techniques and tools. In this chapter, we

[1] An earlier version of this chapter was published in the Proceedings of the 13th IEEE Working Conference on Reverse Engineering (WCRE 2006) (Cleve et al., 2006).

particularly focus on the use of *program slicing techniques* as a means to discover/-validate hypotheses about implicit data structures and constraints. In the DBRE context, program slicing typically helps to reach two intermediate objectives (Henrard, 2003). The first one is the more traditional objective of reducing the visual search space when semi-automatically inspecting source code to discover complex business rules. The second goal concerns the automatic computation of data dependencies. Indeed, discovering dependencies between variables can be used as a basis to retrieve implicit constraints such as undeclared foreign keys. To this end, program slicing provides a very useful intermediate result: the *System Dependency Graph* (SDG).

Program slicing, initially introduced by Weiser (1984), has proved to be a valuable technique to support both software maintenance and reverse engineering processes (Gallagher and Lyle, 1991). However, usual slicing algorithms fail to deal correctly with database application programs. Such programs do not only use standard files, but also store and manipulate data from external Data Management Systems (DMS). Therefore, a traditional slice computed from a database application program will potentially be incomplete or imprecise, due to the data dependencies hidden by the DMS.

In order to improve the accuracy of computed slices, additional data dependencies must be taken into account when constructing the SDG. Those dependencies, induced by the execution of database operations, arise due to the interaction between program variables and database entity types and attributes. In this chapter, we propose an approach to compute accurate program slices in the presence of database statements. We mainly concentrate on the core of standard program slicing techniques, which is the construction of an SDG as precise as possible.

The remainder of this chapter is structured as follows. Section 5.2 defines some basic concepts used in the chapter. In Section 5.3, the nature of the problem to be tackled is described. In Section 5.4, we present a method of computing accurate program slices in the presence of an embedded data manipulation language. Section 5.5 shows how this approach can be extended to handle programs that invoke data access modules to manipulate the database. In Section 5.6 we give an overview of the tools that we developed to support our methodology. Industrial reverse engineering projects are reported in Section 5.7. We discuss related work in Section 5.8. Section 5.9 concludes the chapter and anticipates future work.

5.2 Basic concepts

5.2.1 System dependency graph

Horwitz et al. (1990) introduced a new kind of graph to represent programs, called a system dependency graph, which extends previous dependency representations to incorporate collections of procedures with procedure calls.

The system dependency graph (SDG) for program P is a directed graph whose nodes are connected by several kinds of arcs. The nodes represent assignment

```
    begin
S1    read(A);
S2    if(A < 0)
      then
S3         B := f1(A);
S4         C := g1(A);
      else
S5         if(A = 0)
           then
S6              B := f2(A);
S7              C := g2(A);
           else
S8               B := f3(A);
S9               C := g3(A);
           endif;
      endif;
S10   write(B);
S11   write(C);
    end.
```

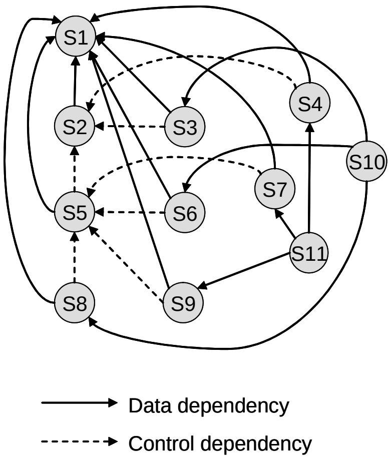


Figure 5.1: A sample program and its corresponding SDG.

statements, control predicates, procedure calls and parameters passed to and from procedures (on the calling side and in the called procedure).

The arcs represent dependencies among program components. An arc represents either a *control dependency* or a *data dependency*. A control dependency arc from node v_2 to node v_1 means that, during execution, v_2 can be executed/evaluated only if v_1 has been executed/evaluated[2]. Intuitively, a data dependency arc from node v_2 to node v_1 means that the state of objects used in v_2 can be defined/changed by the evaluation of v_1. Figure 5.1[3] depicts a small sample program together with its corresponding SDG.

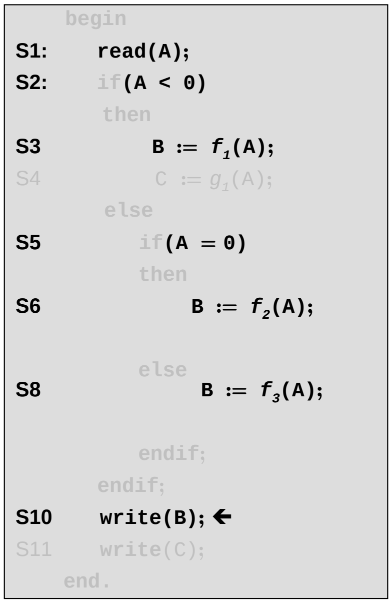
5.2.2 Program slicing

Program slicing (Weiser, 1984), is a well-established technique that can be used to debug programs, maintain programs or understand program behavior (Harman and Hierons, 2001).

The *slice* (or backward slicing) of a program with respect to program point p and variable x consists of all the statements and predicates of the program that may affect the value of x at point p. In Weiser's terminology, a *slice criterion* is a pair $<p,V>$, where p is a program point and V is a subset of the program variables.

[2] The definition is slightly different for calling arcs, but this does not change the principle.

[3] adapted from (Agrawal and Horgan, 1990)



```
     begin
S1:    read(A);
S2:    if(A < 0)
       then
S3         B := f1(A);
S4         C := g1(A);
       else
S5        if(A = 0)
          then
S6            B := f2(A);

          else
S8             B := f3(A);

          endif;
       endif;
S10    write(B);
S11    write(C);
     end.
```

Figure 5.2: Program slice computed on the program of Figure 5.1, using <`S10`,`B`> as criterion.

Figure 5.2 gives the program slice computed on the program of Figure 5.1, using <`S10`,`B`> as criterion.

Horwitz et al. (1990) also proposed an algorithm for interprocedural slicing that uses the system dependency graph. In their approach, a program is represented by an SDG and the slicing problem becomes a node-reachability problem, so that slices can be computed in linear time.

5.2.3 Host variables

Data exchanges between the program and the database is performed by means of so-called *host variables*. We distinguish two kinds of host variables, depending on their role in the database operation:

- *input host variables* are used by programs to pass data to the DMS. For instance, COBOL variables occuring in the `where` clause of an embedded SQL query are of that kind.

- *output host variables* are used by the DMS to pass data and status information to programs. In embedded SQL, typical examples of output host variables are the status variable `SQLCODE`, as well as the COBOL variables occuring in a `into` clause.

5.3 Problem statement

The complexity of the *database-aware* program slicing task lies in the nature of the data manipulation language (DML). We identify four categories, according to the distance between the DML and the *host* programming language, which we assume to be COBOL in the remainder of this chapter.

Native In the case of standard files, programmers use *native* COBOL data access statements (e.g., `READ`, `WRITE`, `DELETE`,...). Since all relevant information is contained in the program, traditional slicing on top of pure COBOL is sufficient.

Built-in For DMS like IDMS/CODASYL, the COBOL language provides *built-in* instructions (i.e., `FIND`, `STORE`, ...) to access the database. Since they are seamlessly incorporated in the COBOL language, such instructions can be analyzed by traditional COBOL slicers as well. The only needed extension is an option to load the definition of the physical schema referenced by the `SUB-SCHEMA SECTION` of the `DATA DIVISION`. This physical schema is obtained through an additional DDL[4] analysis process.

Embedded The third category regroups DMS such as SQL or IMS, for which COBOL does not provide built-in data access statements. To access such DMS programmers write *embedded* instructions in the COBOL program. Such embedded code fragments belong to an external Data Manipulation Language (DML). They are embedded as is in the main programming language, called the host language. The DMS manufacturer provides a pre-processor (precompiler) that translates the embedded DML instructions into COBOL calls to external functions (programs) that implement the access to the DMS. Typical intricacies arise when analyzing programs with embedded DML code:

- The embedded instructions are not standardized and thus vary from one DMS to another;
- The embedded code does not conform to the host language syntax;
- The physical database schema is not explicitly declared in the program itself.

Call-based Many data-intensive programs invoke another program, often called a *Data Access Module* (DAM), in order to access the database. The *Call-based* DML category accounts for this frequent situation. In this case, the slicing problem becomes very complex, especially since the DAM may in turn use a Call-based DML. Furthermore, two additional issues must be considered:

- Unlike embedded DML instructions, the precise behaviour of DAM calls is not explicitly specified in a user manual;

[4] DDL stands for Data Description Language

Native DML	**Built-in DML**

`MOVE 7 TO CUS-ID`
`READ CUSTOMER KEY IS CUS-ID`
`DISPLAY CUS-NAME`
	`MOVE 7 TO CUS-ID`
	`FIND CUSTOMER USING CUS-ID`
	`GET CUSTOMER`
	`DISPLAY CUS-NAME`

Embedded DML	**Call-based DML**

`MOVE 7 TO CUS-ID`
`EXEC SQL`
`   SELECT NAME`
`   INTO :CUS-NAME`
`   FROM CUSTOMER`
`   WHERE ID = :CUS-ID`
`END-EXEC`
`DISPLAY CUS-NAME`
	`MOVE 7 TO CUS-ID`
	`MOVE "CUSTOMER" TO REC-NAME`
	`MOVE "FIND-BY-ID" TO ACTION`
	`CALL "DAM" USING ACTION`
	`                 REC-NAME`
	`                 CUSTOMER`
	`                 STATUS`
	`DISPLAY CUS-NAME`

Figure 5.3: Illustration of native, built-in, embedded, and call-based DMLs

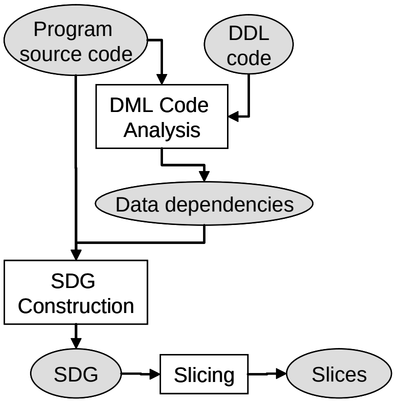


Figure 5.4: Methodology for slicing with embedded DML code

- When analyzing a DAM invocation it is necessary to determine the actual value of each of its input parameters, which is not always statically possible.

Figure 5.3 shows similar COBOL fragments illustrating the differences between the four DML categories. Each code fragment searches for a customer based on a specified identifier and displays its name.

5.4 Slicing with embedded DML

In this section, we develop a general methodology to build system dependency graphs in the presence of embedded DML code. This methodology, depicted in Figure 5.4, includes the analysis of database operations occuring in the program as a first stage. This program analysis phase, discussed in Section 5.4.1, extracts implicit data dependencies from DML code fragments. These dependencies are then used as input to the SDG construction process, as explained in Section 5.4.2.

5.4.1 DML code analysis

The DML code analysis phase consists in analyzing the database operations performed by the program. This process aims at extracting data dependencies between host program variables (input or output) and database variables. Such dependencies are of two possible kinds:

- *direct mappings*

  form: `DIRECT-MAP` var_{in} `TO` var_{out}

  meaning: the value of variable var_{in} *directly* affects the value of variable var_{out}. (var_{in} and var_{out} must be of compatible types). Directly means that there exists a clear function to compute the value of var_{out} from the value of var_{in}.

- *indirect mappings*

  form: `INDIRECT-MAP` $vars_{in}$ `TO` $vars_{out}$ (with $vars_{in}$ being optional)

  meaning: the values of variables $vars_{out}$ are *indirectly* affected by the values of variables $vars_{in}$. Indirectly means that the way the values of $vars_{in}$ influence the values of $vars_{out}$ is less precise.

As an illustration, let us consider the COBOL/SQL program fragment shown in Figure 5.5. COBOL instructions and host variables are in upper-case, while embedded SQL code is in lowercase. The program fragment displays the name and the phone number of each customer living in a given city. It first accepts a zip code from the command-line, opens a SQL cursor and fetches each of its rows.

The `open` statement (lines 52-54) involves two different kinds of data dependencies. First, the value of the input host variable `CUS-ZIP` *directly* affects the value of column `zip`. Second, the value of column `zip` has an *indirect* influence on the values of the columns occuring in the `select` clause (`name` and `phone`). In addition, the value of status variable `SQLCODE` is also affected. Thus we can extract the following mapping statements from the embedded `open` statement:

```
DIRECT-MAP CUS-ZIP TO zip
INDIRECT-MAP zip TO name phone SQLCODE
```

When analyzing the `fetch` query (lines 79-83), implicit data dependencies can be detected. First, fetching a cursor indirectly influences the value of the status variable `SQLCODE`. Second, the values of the output host variables `CUS-NAME` and `CUS-PHONE` are *directly* affected by the values of the corresponding selected columns (`name` and `phone`). These implicit dependencies can be formalized by the following dependency pseudo-instructions:

```
INDIRECT-MAP TO SQLCODE
DIRECT-MAP name TO CUS-NAME
DIRECT-MAP phone TO CUS-PHONE
```

```
exec sql
  declare byzip
  cursor for
  select name,
        phone
  from customer
  where zip = :CUS-ZIP
  order by name
end-exec.
...
ACCEPT CUS-ZIP
exec sql
 open byzip
end-exec.
MOVE 0 TO END-SEQ.
PERFORM DISPLAY-CUS
 UNTIL END-SEQ = 1.
...
DISPLAY-CUS.
exec sql
  fetch byzip
  into :CUS-NAME,
       :CUS-PHONE
end-exec.
IF SQLCODE NOT = 0
   MOVE 1 TO END-SEQ
ELSE
   DISPLAY CUS-NAME "-"
           CUS-PHONE.
```

Figure 5.5: A COBOL/SQL code fragment

Let us now express the general rule that can be used to extract dataflow dependencies from embedded SQL code. This necessitates some definitions. Let q be an embedded SQL query:

- *in-couple*(q) is the list of the couples (hv, c), such that hv is an input host variable used in q and c is the column associated with hv in q.

- *out-couple*(q) is the list of the couples (hv, c), such that hv is an output host variable used in q and c is the column associated with hv in q.

- *columns*(*list-couples*) is the list of the columns occuring in *list-couples*, a list of couples (hv, c), such as hv is a host variable and c is a column.

- *out-hv*(q) is the list of all the output host variables used in q and that do not appear in *out-couple*(q). In other words, that list contains the output host variables that are not directly associated with a column, such as the status variable `SQLCODE` or variables resulting from aggregation queries.

Having the above definitions in mind, we can specify the general rule used to extract dataflow dependencies from an arbitrary embedded SQL query q as follows:

```
for all $(hv, c) \in$ in-couple$(q)$ : extract(DIRECT-MAP $hv$ TO $c$)
extract(INDIRECT-MAP columns(in-couple$(q)$) TO columns(out-couple$(q)$) out-hv$(q)$)
for all $(hv, c) \in$ out-couple$(q)$ : extract(DIRECT-MAP $hv$ TO $c$)
```

The first part of the rule derives direct mappings between input host variables and their corresponding SQL columns. Those mappings typically occur in the `where` clause of the query. The second part extracts the indirect mapping from the columns associated with input host variables to the columns associated with output host variables and the output host variables that are not associated with any column. The last part of the rule identifies the direct mappings between the selected columns and their corresponding output host variables. Note that, since both *in-couple* and *out-couple* lists may be empty, the minimal data dependency that can be extracted from an embedded SQL query is the following:

```
INDIRECT-MAP TO SQLCODE
```

This data dependency means that the execution of the query indirectly influences the value of `SQLCODE`.

5.4.2 SDG construction

The first step for computing program slices is the construction of the SDG. In the presence of embedded DML statements, this SDG must represent the control and the dataflow of both the host language and the embedded language. In order to produce the SDG, the results of the DML code analysis process are exploited. The dependency pseudo-instructions (`DIRECT-MAP` and `INDIRECT-MAP`) are used (instead of the original code) to construct the SDG nodes and the data dependency arcs corresponding to embedded DML fragments. All SDG nodes remain linked to the initial source code locations.

This technique allows the SDG construction process to become DML-independent. The only language to be considered is the host language (here COBOL), augmented with the two additional pseudo-instructions. This ad-hoc grammar augmentation is inspired by the *scaffolding* technique proposed by Sellink and Verhoef (2000).

Once the full SDG has been built, program slices can be computed using the usual algorithm.

5.5 Slicing with call-based DML

In this section, we show how our methodology, depicted in Figure 5.4, can be extended to program slicing in the presence of a data access module (DAM). In other words, we will show how we can adapt our approach so that we can correctly analyse programs that use a call-based DML.

We can distinguish three possible kinds of DAM-based data access:

- **Global DAM**: there is only one DAM used to access all the tables/record types and to perform all the possible actions (read, write,...);

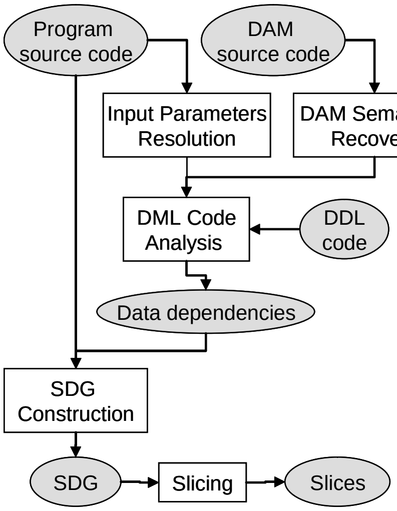


Figure 5.6: Methodology to slice programs with call-based DML

- **One DAM per table/record type**: each table/record type has a dedicated DAM, which implements all the possible actions;
- **One DAM per action**: each action has a dedicated DAM, which can be called to access all the tables/record types.

Our approach to SDG construction with DAM invocations (depicted in Figure 5.6) extends the methodology described in Section 5.4. The extension is based on the following principles:

- Any data access module can be seen as a data management system;
- Any data access module invocation can be regarded, and analyzed, as a DML instruction.

However, analyzing DAM invocations necessitates some additional knowledge on the DAM behavior itself. As already indicated, we cannot assume that such a knowledge is explicitly available. Consequently, our methodology requires an extra step that consists in recovering a deep understanding of the behaviour of the DAM. One can consider the DAM semantics as a correspondence table, which identifies the DML instruction(s) actually involved in each possible kind of DAM invocation. This leads us to another issue, namely *input parameter resolution*. In order to make the link between a DAM invocation and its corresponding DML instructions, one also need to determine the actual value of each input parameter of this invocation.

Slicing in the presence of DAM invocations might lead to very imprecise results. Indeed, we do not always know the actual value of the parameters used to call a DAM. Typically, the table/record type to access and the action to be performed

```
203   ACCEPT AN-ID.
204   MOVE AN-ID TO CUS-ID.
205   MOVE "read" TO ACTION.
206   MOVE "CUSTOMER" TO REC-NAME.
207   CALL "DAM"
208      USING ACTION
209            REC-NAME
210            CUSTOMER
211            STATUS.
212   IF (STATUS-OK)
213     DISPLAY CUS-NAME CUS-PHONE
214   ELSE
215     DISPLAY "UNKNOWN CUSTOMER".
```

Figure 5.7: Calling program code

are unknown. In this situation, the computed slices will typically contain a lot of noise (in the worst case, all the possible execution paths of the DAM).

Let us consider the code fragment shown in Figure 5.7. In this example, a program invokes a data access module in order to retrieve a customer record based on the customer identifier. The DAM code is provided in Figure 5.8. If the customer is found, the calling program displays its name and phone number. In this example, if the slice w.r.t. the `DISPLAY` instruction (line 213 of Figure 5.7) was computed, it would include both select instructions (lines 956 and 1125 of Figure 5.8), since the static slicer cannot make any assumption about the value of input variables `REC-NAME` and `ACTION`. Replacing DAM invocations with corresponding DML instructions can significantly improve the accuracy of computed slices. Indeed, this may allow the SDG construction process to consider *relevant* control and data flows only. For instance, if one replaced the DAM call (lines 207-211 of Figure 5.7) with the corresponding select clause of Figure 5.8 (lines 955-960), then the slice w.r.t. the `DISPLAY` instruction would become minimal.

5.5.1 DAM semantics recovery

Analyzing a DAM invocation obviously calls for a correct understanding of the way the DAM is implemented. This requires, among others, to determine the meaning of the different DAM call parameters. The name of the DAM and the value of the input parameters are usually sufficient to translate a DAM invocation into corresponding DML instructions. Unfortunately, no universal method exists for recovering the semantics of the parameters. As usual in program understanding, this typically relies on the combination of (partial) program analysis results, domain knowledge, DAM construction knowledge and user/programmer interviews.

In our example of Figure 5.7, the program invokes the data access module using four parameters. Through the DAM semantics recovery phase, we can learn that:

- the first parameter (`ACTION`) specifies the database operation to be performed (read, write, etc.);

```
IF (ACTION = "read")
    IF REC-NAME = "CUSTOMER"
       PERFORM READ-CUSTOMER
    END-IF
    IF REC-NAME = "ORDERS"
       PERFORM READ-ORDERS
    END-IF
    ...
END-IF
...
READ-CUSTOMER.
 exec sql
   select name, phone
   into :CUS-NAME, :CUS-PHONE
   from customer
   where id = :CUS-ID
 end-exec.
 MOVE SQLCODE TO STATUS.
 ...
READ-ORDERS.
 exec sql
   select ...
   from orders
   ...
 end-exec.
 ...
```

Figure 5.8: Data access module code

- the second parameter (`REC-NAME`) contains the name of the record type to be accessed;
- the third parameter (`CUSTOMER`) is the resulting record itself;
- the last parameter (`STATUS`) is an output status variable.

5.5.2 Input parameters resolution

In order to correctly analyze a DAM invocation, one also needs to determine the actual value of each input argument (here, `ACTION` and `REC-NAME`). This process can be supported by a customized backward slicing phase, searching for predecessor instructions that initialize DAM input parameters.

In our example, the `MOVE` statement of line 205 (resp. 206) will be found and analyzed, to determine the actual value of `ACTION` (resp. `REC-NAME`) at the time of calling the DAM.

5.5.3 DAM Call Analysis

Figure 5.9 summarizes the successive steps that may be needed when analyzing DAM invocations. First, input parameters are resolved. Second, the DAM call is replaced by its corresponding DAM code fragment, including the actual DML instructions. Finally, the DML code is analyzed as described in Section 5.4.1. The

Original Code	CALL "DAM" USING ACTION REC-NAME CUSTOMER STATUS
Parameters Resolution	CALL "DAM" USING "read" "CUSTOMER" CUSTOMER STATUS
DAM Semantics Recovery	exec sql select name, phone into :CUS-NAME, :CUS-PHONE from customer where id = :CUS-ID end-exec. MOVE SQLCODE TO STATUS.
DML Code Analysis	DIRECT-MAP CUS-ID TO id INDIRECT-MAP id TO name phone SQLCODE DIRECT-MAP name TO CUS-NAME DIRECT-MAP phone TO CUS-PHONE MOVE SQLCODE TO STATUS.

Figure 5.9: Successive steps to analyze the DAM call of Figure 5.7

```
[select-into-analysis]
dataflow-analysis(exec sql at-db-clause? select-stat end-exec)
= gen-direct-map-hv2c(in-couple)
  INDIRECT-MAP columns(in-couple) TO columns(out-couple) SQLCODE
  gen-direct-map-c2hv(out-couple)
when in-couple  :=  in-couple(select-stat),
     out-couple :=  out-couple(select-stat)
```

Figure 5.10: A sample ASF equation for embedded SQL analysis

resulting code/pseudo-code will be used, instead of the original DAM invocation, to build the SDG.

5.6 Tool support

5.6.1 Program analysis

We implemented DML code analyzers for embedded SQL and call-based IMS. Both analyzers rely on the ASF+SDF Meta-Environment (van den Brand et al., 2001). We reused an SDF version of the IBM-VSII COBOL grammar, which was obtained by Lämmel and Verhoef (2001). We wrote SDF modules specifying (a sufficient subset of) the syntax of embedded SQL (resp. IMS calls). On top of this augmented COBOL grammar, we implemented a set of ASF equations (i.e., rewrite rules), that analyse DML fragments and accumulate implicit data dependencies.

```
[get-unique-nalysis]
dataflow-analysis(CALL 'CBLTDLI' USING GET-UNIQUE param1 segment params ssa,
                  ssa-segment-tbl, ssa-field-tbl, ssa-value-tbl)
=  DIRECT-MAP value TO field
   INDIRECT-MAP field TO segment
when field := lookup(ssa-field-tbl,ssa),
     value := lookup(ssa-value-tbl,ssa)
```

Figure 5.11: A sample ASF equation for IMS calls analysis

Figures 5.10 and 5.11 provide examples of such rewrite rules. The rewrite rule of Figure 5.10 analyzes a `select` query and extracts its direct and indirect data dependencies. It implements the general rule specified in Section 5.4 in the particular case of a `select` statement. The rule can be applied to the example shown in Figure 5.9, for which the value of `in-couple` is `[<CUS-ID,id>]`, while the value of `out-couple` is `[<CUS-NAME,name>,<CUS-PHONE,phone>]`. Figure 5.11 shows an example rewrite rule for analyzing a `GET UNIQUE` IMS call. In this case, the dataflow analyzer takes three correspondence tables as additional arguments. Those additional arguments correspond to the result of the *DAM semantics recovery* step. They allow the rewrite rules to retrieve, for each *segment search argument* (ssa) (1) the name of the associated IMS segment[5], (2) the name of the reference field and (3) the name of the field containing the reference value for the search.

The result of the dataflow analysis process consists of a set of lines. Each line represents a single data dependency extracted from a given DML fragment, and provides the following information:

- full path name of the program;
- source code location of the DML fragment (file name, begin and end line numbers);
- dependency sequence number in the fragment;
- kind of mapping (direct or indirect);
- input variables (host variables or database fields);
- output variables (host variables or database fields).

The dataflow dependencies extracted from the code are then loaded in a relational database, which is taken as an input by the SDG construction process.

5.6.2 SDG construction and slicing

The program slicing tool (Henrard, 2003) analyzes COBOL programs with procedures (`PERFORM`), sub-routines calls (`CALL`) and arbitrary control flows (`GO TO`). A

[5] a record type is called a *segment* in IMS

program is represented by a graph (the system dependency graph) and the slicing problem is simply a vertex-reachability problem. Therefore, slices may be computed in linear time in the number of edges when the graph has already been computed.

The construction of the graph is much more costly. For the interprocedural slice, we use the SDG to represent the program and the algorithm that was described by Horwitz et al. (1990) to compute the slice. This algorithm can handle procedures and sub-routines.

The SDG construction algorithm can only manipulate variables that are local to procedures but cannot handle global variables present in COBOL programs. Therefore, for each procedure, we recover all the variables used (referenced and modified) by the procedure and create corresponding formal-in and formal-out parameters.

We use the augmented system dependency graph as proposed by Ball and Horwitz (1992) to solve the orthogonal problem of slicing procedures with arbitrary control flows. The variables are represented by their physical position and their length. Indeed, we cannot use the names of the variables because they can be made of sub-level variables and can be *redefined*.

Both the SDG construction and the slicing algorithm are implemented in C++.

5.6.3 Parameter resolution

During the construction of the SDG, both the variables used (referenced and modified) by each instruction and the constants used are stored. This allows to automate the resolution of DAM input parameters by querying the SDG for the possible values of each input parameter of a DAM invocation. To determine which values (constants) can be stored in an input parameter, the dataflow edges of the SDG are followed backwards starting form the DAM call instruction until a constant is reached. If the parameter resolution process identifies *several* possible values for the same input parameter at the same program point, then a *set* of corresponding DML statements are considered when building the SDG.

5.7 Industrial applications

The approach and tools described in this chapter have been used for supporting several industrial reverse engineering projects. Three of them are presented in this section.

5.7.1 COBOL/embedded SQL

The first project consisted of the reverse engineering of a relational database for a Belgian insurance company. The company planned the migration of its information system towards a new database platform, and needed to evaluate the complexity and cost of the database migration process. In this context, our goal was to demonstrate that automated program analysis techniques could allow to recover relevant data dependencies in the database schema.

The original relational schema included 27 tables and 784 columns, but did not declare any explicit foreign key. The implicit knowledge of the database integrity constraints was almost completely lost.

Our methodology and tools were applied to a small subset of the application consisting of 95 programs and totalizing 150 thousands lines of COBOL code. The embedded SQL fragments include 112 cursor declarations, 114 open statements, 114 fetch statements, 17 delete queries, 38 insert queries and 41 update queries.

The analysis of the embedded SQL code resulted in the extraction more than 5000 implicit dependencies. The construction and analysis of the SDG allowed to derive the following information:

- the set of tables (only 15) actually used by the 95 programs;
- the set of colums of these tables accessed by each program;
- 32 implicit data dependencies between columns, that proved to be undeclared foreign keys.

This project confirmed that the analysis of embedded SQL fragments may help to recover implicit referencial constraints between tables based on the detection of (indirect) dataflows between columns of distinct tables. From such dataflows we can derive foreign key candidates, to be confirmed through a validation phase.

5.7.2 COBOL/call-based CODASYL

The second project concerned the reverse engineering of a large COBOL system for a Belgian public administration. The system was built on top of an IDS/II (CODASYL) database comprising 232 record types, 150 set types[6] and 648 fields. The analyzed application consisted of 2287 programs, totalizing more than 2 million lines of code.

In this application, database operations were performed through a single data access module, providing reading and writing access to all the record types. Each DAM invocation should specify as parameters (1) the corresponding record name, (2) a variable to store the record and (3) the type of access to be performed (`get first`, `get next`, `store`, ...).

The DAM also computed a physical position (`DB-KEY`) before inserting a new record in the database. This physical position served as a basis for storing records in different *areas* (files) according to a complicated rule aiming to optimize data access speed.

Thanks to the analysis of the DAM invocations occuring in the programs (5952 COBOL `CALL` statements) it was possible to:

[6] In CODASYL, one-to-many relationship types are called *set types*, as we will see in Chapter 8.

- draw the usage graph, specifying which program uses which record type;
- enrich each record type with a finer-grained structural decomposition;
- discover more than 2000 implicit data dependencies.

In this particular project, the resolution of the input parameters of the DAM calls proved essential for building a precise usage graph. Without this step, it would have been impossible to determine which record is accessed nor which operation is performed. Indeed, the program slice computed from any DAM invocation (without resolving its parameters) would have contained all read and write accesses to each record type.

Regarding schema enrichment, the dependencies extracted from the DAM calls also revealed very usefull. Originally, each record type definition in the DDL code consisted of only two declared fields: (1) the access key to the record and (2) a second large field containing the remainder of the data. In contrast, the COBOL variable used to store the record (output argument of DAM invocations) exhibited a much more precise decomposition than the corresponding record type.

5.7.3 COBOL/call-based IMS

The third industrial project aimed at redocumenting a hierarchical database (IMS) for an american private company. This project made use, among others, of our dataflow analysis approach for call-based DML statements.

The physical schema of the IMS database comprised 40 segment types and 107 fields. This schema was quite obsure and imprecise. First, the fields declared in the DDL code were mainly those used as indexes. Second, the physical names for segment types and fields were limited to eight characters, which made them hardly understandable.

The system to analyse consisted of 226 COBOL programs (500 000 lines of code), accessing the IMS database in a call-based manner. Through the automated analysis of 1488 database calls, we extracted the set of COBOL variables used to manipulate each IMS segment. The precise declarations of the identified variables allowed us to significantly enrich the physical schema with more meaningfull names and finer-grained segment type decompositions.

5.7.4 Lessons learned

The industrial projects presented above clearly demonstrate that it is possible to construct accurate SDGs from programs involving database operations, especially when database access is performed using an extension of the programming language (embedded SQL) or through a data access module.

Our approach is based on the *divide and conquer* principle. It consists in analyzing separately the database manipulation fragments, and then in using the results of this analysis, instead of the original fragments, for producing the SDG. From that point it is possible to use traditional SDG querying or program slicing.

One important lesson is thus it may be necessary to analyse/pre-process a program (leading to an intermediate, incomplete SDG), in order to construct, in a second stage, a complete and accurate SDG. This is especially true in the presence of data access module invocations, for which input parameter resolution is often required.

Experience also confirmed that every reverse engineering project is different from other ones. Hence the need for programmable, extensible and customizable tools. In this context, the reusability of our slicing approach and tools appears as an important advantage. For instance, when programs based on a new embedded language must be analyzed, the main necessary adaptation concerns the dependency analysis rules. The SDG construction and the slicing algorithm remain unchanged. Our tool-supported approach is also capable to deal with programs that use several embedded languages or that use embedded languages in combination with one or several data access module.

5.8 Related work

5.8.1 Previous work

Program slicing has long been considered as a valuable technique to support various software maintenance tasks such as debugging, regression testing, program understanding, and reverse engineering (Gallagher and Lyle, 1991; Beck and Eichmann, 1993). A lot of researchers have extended the SDG to represent various language features and proposed variations of dependence graphs that allow finer-grained slicing. Among them, we mention the work by Agrawal (1994) and Jackson and Rollins (1994).

Tan and Ling (1998) recognise that traditional slicing methods cannot correctly deal with programs involving database operations. To face this limitation, they suggest the introduction of implicit pseudo-variables to capture the influence among I/O statements that operate on COBOL files. For each COBOL record type, a pseudo-variable is assumed to exist and to be updated when the file access statements are executed. Such pseudo-variables allow to introduce additional data dependencies at the record level.

More recently, Willmor et al. (2004) propose a novel approach to program slicing in the presence of database states, which considers new forms of data dependencies. The first category, called *program-database* dependencies, accounts for interactions between program statements and database statements. The *database-database* dependencies capture the situation in which the execution of a database statement affects the behaviour of another database statement.

As already indicated, the way we simulate the dataflow behavior of DML code with pseudo-instructions is similar to the scaffolding technique proposed by Sellink and Verhoef (2000). Within the context of software renovation, source-code scaffolding consists in inserting some markup in the source code in order to store intermediate results of analysis/transformation processes and to share information between tools. In our case, we do not transform or scaffold the source code itself, but we store extracted dependencies into an external database.

5.8.2 Discussion

As Tan and Ling, we consider additional data dependencies involved in the execution of database operations. But our approach aims at extracting variable dependencies at a finer-grained level. For instance, when analyzing an embedded SQL `update` statement, it is not sufficient to conclude that a table is updated. One must also determine the set of columns that are actually affected and, if relevant, one should make the link between these columns and the corresponding COBOL input host variables.

Our approach complements the work by Willmor et al. (2004). Indeed, we focus on recovering dependencies between program and database *variables*, while they consider additional dependencies between program and database *statements*. It is obvious that the data dependencies our tools automatically extract from database statements allow, in a second stage, to compute inter-statement dependencies in the style of Willmor et al. (2004). Our dataflow dependency results allow to determine the set of variables that are used and defined by each database statement. In addition, those dependencies capture the implicit links that hold between program variables and corresponding database variables. Another important difference lies on the fact that we clearly separate the analysis of the database operations from the SDG construction phase. This allows the latter to become DML-independent, and thus increases its reusability.

This chapter also constitutes an extension of Jean Henrard's PhD thesis (Henrard, 2003), that proposes slicing-based program analysis techniques and tools in support to database reverse engineering. In the same thesis, a set of dataflow dependency extraction rules for embedded SQL are already suggested. The major contributions of this chapter concern (1) the automation of the latter rules resulting in a dataflow analyser for embedded SQL, (2) the extension of the methodology to call-based DML statements analysis, (3) the developpement of a dataflow analyser for call-based IMS, and (4) the application of the methods and tools to large-scale legacy systems.

5.9 Conclusions

In this chapter, we presented a general methodology that allows to compute accurate system dependency graphs in the presence of embedded database operations. The methodology is based on the combination of embedded code analysis and DML-independent SDG construction. We showed how this methodology can be generalized to the analysis of programs invoking data access modules (DAM).

While industrial reverse engineering projects have shown the suitability and usefullness of our dependency analysis approach and tools, precisely measuring the positive effect of our methodology on the accuracy of constructed SDGs still remains to be done. Unfortunately, such an evaluation proved particularly complicated in an industrial context. Indeed, case studies are very costly and require the cooperation of a customer ready to invest in comparative experiments.

We anticipate several directions for future work in *database-aware* dataflow analysis. In particular, we would like to explore the use of SDG querying techniques for database applications reengineering and evolution. For instance, SDG analysis appears as a promising candidate technique for *understanding* the data access logic of a program, before adapting it to an evolving database schema or platform[7].

Roadmap

This chapter described a dataflow analysis approach for database queries relying on *static program analysis*, thus taking as input the *source code* of the programs. Obviously, this approach may fall short in the presence of *dynamically generated* database queries. Chapter 6 will attempt to partially face this limitation, by exploring the use of *dynamic program analysis* techniques for SQL queries. In the latter case, query analysis is based on the *execution trace* of the application programs. In contrast with the present chapter, which mainly focused on *intra-query* dependency analysis, the next chapter will also take *inter-query* dependencies into account.

[7] in the style of the *Logic Rewriting* program conversion strategy (P3) presented in Chapter 4

Chapter 6

Dynamic Analysis of SQL Queries

The most exciting phrase to hear in science, the one that heralds new discoveries, is not 'Eureka!' but 'That's funny...'
– Isaac Asimov

This chapter[1] explores the use of dynamic analysis techniques for analyzing SQL statements in the context of database reverse engineeering. It identifies, illustrates and compares possible techniques for (1) capturing SQL query executions at run time and (2) extracting implicit schema constructs from SQL query execution traces. Finally, it discusses the usefullness of such techniques based on an initial experiment.

6.1 Introduction

Data-intensive systems exhibit an interesting symmetrical property due to the mutual dependency of the database and the programs. When no useful documentation is available, it appears that (1) understanding the database schema is necessary to understand the procedural code and, conversely, (2) understanding what the procedural code is doing on the data considerably helps in understanding the properties of the data.

Procedural code analysis has long been considered a complex but rich information source to redocument database schemas. Even in ancient programs based on standard file data managers, identifying and analysing the code sections devoted to the validation of data before storing them in a file allows developers to detect constructs as important as actual record decomposition, uniqueness constraints, referential integrity or enumerated field domains. In addition, navigation patterns in source code can help identify such important constructs as semantic associations between record types.

[1] A short version of this chapter was published in the Proceedings of the 15th IEEE Working Conference on Reverse Engineering (WCRE 2008) (Cleve and Hainaut, 2008).

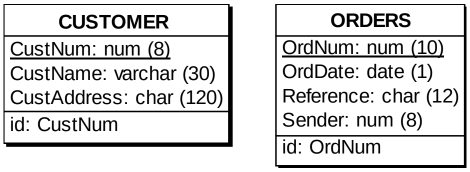


Figure 6.1: Two tables including implicit constructs

When, as has been common for more than two decades, data are managed by relational DBMSs, the database/program interactions are performed through the SQL language and protocols. Based on the relational algebra and the relational calculus, SQL is a high-level language that allows programmers to describe in a declarative way the properties of the data they instruct the DBMS to provide them with.

By contrast, navigational DMLs (also called *one-record-at-a-time* DML) access the data through procedural code that specifies the successive operations necessary to get these data. Therefore, a single SQL statement can be the declarative equivalent of a procedural section of several hundreds of lines of code (LoC). Understanding the semantics of an SQL statement is often much easier than that of this procedural fragment. The analysis of SQL statements in application programs is a major program understanding technique in database reverse engineering (Petit et al., 1994; Andersson, 1998; Embury and Shao, 2001; Willmor et al., 2004; Cleve et al., 2006).

6.1.1 SQL code analysis

We illustrate the importance of SQL statement analysis on a small but representative example based on the schema of Figure 6.1 made up of two tables describing customers and orders. This schema graphically translates the constructs of the DDL code. Query 1 obviously extracts the customer city and the ordered product for a definite order. It asks the DBMS to extract these data from the row built by

```
select substring(CustAddress from 61 for 30), Reference
into   :CITY, :PRODUCT
from   CUSTOMER C, ORDERS O
where  C.CustNum = O.Sender
and    OrdNum = :ORDID
```

Query 1. Extracting hidden City and Product information (predicative join)

joining tables `CUSTOMER` and `ORDERS` for that order. It exhibits the main features of the program/query interface: the program transmits an input value through host variable `ORDID` and the query transmits result values in host variables `CITY` and `PRODUCT`. The analysis of this query brings to light some important hidden information, two of which are essential for the understanding of the database schema.

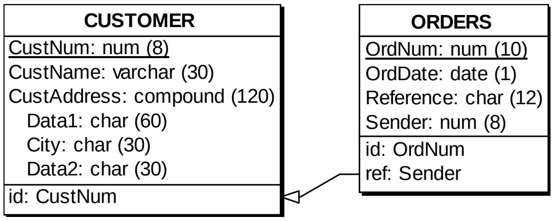


Figure 6.2: Two implicit constructs revealed by the analysis of Query 1

1. The join is performed on columns `CustNum` and `Sender`. The former is the primary key (the main identifying column) of table `CUSTOMER` while the second one plays no role so far. Now, we know that most joins found in application programs are based on the matching of a foreign key (a column that is used to reference a row in another table) and a primary key. As a consequence, Query 1 strongly suggests that column `Sender` is a foreign key to `CUSTOMER`. Further analysis will confirm or reject this hypothesis.

2. The seemingly atomic column `CustAddress` appears to actually be a compound field, since its substring at positions 61 to 90 is extracted and stored into a variable named `CITY`[2].

Translating this new knowledge in the original schema leads to the more precise schema of Figure 6.2.

The SQL code of Query 1 *explicitly* exhibits the input and output variables through which the host statements interact with the query. These variables are important since, on the one hand, input variables define the variable parts of the query and, on the other hand, input and output variables potentially weave links with other SQL queries. The code fragment Query 2, that develops the same join as that of Query 1 in a procedural way, illustrates such an inter-query dependency. Naming *Q21* and *Q22* the two fragments, we observe that the host variable `CUST` is the output variable of query *Q21* and the input variable of the query *Q22*. If the intermediate host statements do not change the value of `CUST` (this invariance can be checked through a dependency analysis), then these queries can be considered

```
select Sender into :CUST
from   ORDERS where  OrdNum = :ORDID
<host statements>
select CustName, CustAddress into :CNAME, :CADDRESS
from   CUSTOMER where  C.CustNum = :CUST
```

Query 2. Extracting hidden City and Product information (procedural join)

[2] In practice, it might happen that several variables with different names are used to store the value of the same column fragment. In this case, the analyst is in charge of choosing the most meaningful name for the new sub-level attribute.

as a global query. Such inter-query analysis has been suggested by Petit et al. (1994) for example.

The analysis of SQL statements in a program can address each statement independently (Query 1) or can extend the understanding to chains of dependent statements (Query 2).

6.1.2 Static vs dynamic SQL

These introductory examples are expressed in *static SQL*, a variant of the language in which the SQL statements are hard-coded in the source program. There is another family of SQL interfaces, called *dynamic SQL*, with which the SQL statements are built at runtime and sent to the database server through a specific API. Typically, these programs *build* each query as a character string, then ask the DBMS to *prepare* the query (i.e., to compile it) and finally *execute* it. The only point in the program at which the actual query exists is at runtime, when, or just before, the query string is sent to the DBMS for compilation and/or execution. Query 3 is a simple example of the use of dynamic SQL to extract the name of customer 'C400'.

```
CNUM = "C400";
QUERY = "select CustName from CUSTOMER where CustNum = ’" + CNUM + "’";
exec SQL prepare Q from :QUERY;
exec SQL execute Q into :NAME;
```

Query 3. First example of dynamic SQL

6.1.3 Static vs dynamic analysis of (dynamic) SQL statements

With some disciplined programming styles (Query 3 is an example), the mere examination of the code of the program provides enough information to infer the actual content of the query string before execution. However, the way this string is computed can be so complex and tricky that only runtime analysis can yield this value. Capturing, saving and processing the values of the query string at runtime resorts to *dynamic program analysis*.

6.1.4 Structure of the chapter

The objective of this chapter is to identify, study and apply dynamic analysis techniques of static and dynamic SQL statements, in the particular context of database reverse engineering. Section 6.2 describes in further detail the dynamic SQL interface as well as the scope of its static and dynamic analyses. Section 6.3 identifies important objectives and applications to which dynamic analysis of SQL can contribute significantly. Section 6.4 identifies and describes ten different techniques for capturing SQL statements traces, a comparative evaluation of which is given in Section 6.5. Section 6.6 particularly focuses on the use of aspect-based dynamic analysis of SQL statements. Technical aspects of SQL trace processing are discussed in Section 6.7. In Section 6.8, we show how SQL traces can be analyzed in the context of database reverse engineering. Section 6.9 presents an initial experiment aiming to evaluate the suitability of SQL trace analysis for detecting implicit referential constraints. Concluding remarks are given in Section 6.10.

6.2 Static VS dynamic SQL

The introductory examples of Queries 1 and 2 are expressed in *static SQL*, which is a variant of the language in which the SQL statements are hard-coded in the source program. The code of static SQL explicitly shows the architecture of the query according to the SQL native syntax. It mentions the schema constructs concerned and identifies the input and output variables through which the host statements interact with the query. Static SQL will appear in standard languages under the name *embedded SQL* (ESQL) and in Java as SQLJ. Many proprietary database languages, such as InterBase, Oracle (PL/SQL) or Sybase and SQL Server (Transact SQL) are based on static SQL as well, though some of them provide some form of dynamicity.

6.2.1 Dynamic SQL

Dynamic SQL or call-level interface (CLI), that has been standardized in the eighties and implemented by most relational DBMS vendors, is illustrated in Queries 3 and 4. The most popular DBMS-independent APIs are ODBC, proposed by Microsoft, and JDBC, proposed by SUN. Dynamic SQL provides a high level of flexibility but the application programs that use it may be difficult to analyse and to understand. Most major DBMS, such as Oracle and DB2, include interfaces for both static and dynamic SQL.

```
QUERY = "select CustName from CUSTOMER where CustNum = :v1";
exec SQL prepare Q from :QUERY;
exec SQL execute Q into :NAME using :CNUM;
```

Query 4. Another common usage pattern of dynamic SQL

The examples in Queries 3 and 4 are written in dynamic SQL for C/C++. The first one shows the build time injection of constants from variable `CNUM`, while Query 4 illustrates the binding of formal variable `v1` with the actual host variable `CNUM` at execution time.

The ODBC and JDBC interfaces provide several query patterns, differing notably on the binding technique, that we refer to in this study. The most general form is illustrated in Query 5, where the iteration structure for obtaining results has been ignored for simplicity. Line 1 creates database connection `con`. Line 2 builds the SQL query in host variable `SQLquery`. This statement includes input placeholder `?` which will be bound to an actual value before execution. Line 3 creates and prepares statement `SQLstmt` from string `SQLquery`. This statement in completed in Line 4, by which the first (and unique) placeholder is replaced with value C400. The statement can then be executed (Line 5), which creates the set of rows `rset`. Method `next` of `rset` positions its cursor on the first row (Line 6) while Line 7 extracts the first (and unique) output value specified in the query select list and stores it in host variable `Num`. Line 8 closes the result set.

```
    CNum = "C400";
1   Connection con = DriverManager.getConnection(DBurl, Login, Password);
2   String SQLquery = "select OrdNum from ORDERS where Sender = ?";
3   PreparedStatement SQLstmt = con.prepareStatement(SQLquery);
4   SQLstmt.setString(1, CNum);
5   ResultSet rset = SQLstmt.executeQuery();
6   rset.next();
7   Num = rset.getInt(1)
8   rset.close();
```

Query 5. Standard JDBC database interaction

When input binding is performed by build time value injection, the prepare and execute steps can be merged, as illustrated in Query 6, which is equivalent to Query 5 (operations of Lines 1 and 8 omitted).

```
String SQLquery = "select OrdNum from ORDERS where Sender = ’"+"C400"+"’";
Statement SQLstmt = con.createStatement();
ResultSet rset = SQLstmt.executeQuery(SQLquery);
rset.next();
Num = rset.getInt(1)
```

Query 6. Concise JDBC database interaction

6.2.2 Static vs dynamic analysis of dynamic SQL statements

In some disciplined programming styles, the building step is written as a sequence of explicit substring concatenations just before the prepare/execute section, so that significant fragments of the SQL query, if not the whole query itself, can be recovered through careful static analysis (van den Brink et al., 2007) or symbolic execution (Ngo and Tan, 2008). However, some fragments of the query may be initialized long before, and far away from the execution point, or computed, or extracted from external sources (file, database, user interface, web pages, etc.) in such a way that discovering the intended query by code examination alone may prove impossible. The JDBC fragment of Query 7 illustrates the problem. There obviously is no realistic static analysis procedure that could provide us with the actual SQL queries that will be executed. The only way to know the actual query is to capture it at runtime, when the `executeQuery` method is executed.

```
String query, SQLv, SQLa, SQLs, SQLc;
SQLv = currentAction; SQLa = keyboard.readString();
SQLs = userDefaultTable; SQLc = getFilter(currentDate, filterNumber);
Connection con = DriverManager.getConnection(url, login, pwd);
Statement stmt = con.createStatement();
query = SQLv + SQLa + SQLs + SQLc;
ResultSet rset = SQLstmt.executeQuery(query);
```

Query 7. An example of dynamic SQL (JDBC)

6.2.3 Capturing results of SQL statement execution

Dynamic analysis has been considered so far as a means to get the actual SQL code at execution time in order to examine it. This technique can also be used to examine the *results* of SQL query executions. The information is captured after code execution. The format of the output data is defined by the form of the SQL query and the information of the logical schema. Result analysis often is the only technique to trace the link between two SQL queries through shared host variables when the intermediate host statements cannot be analysed statically (see Query 2 for example).

6.2.4 Dynamic SQL patterns

Any dynamic statement can be seen as a partially filled frame. Some parts are constant and will appear at the same place in all the instances of the statement while the other parts are drawn from variables or computed to complete the frame. There are two extreme cases: all the parts of the frame are variable and the whole frame is a pure constant. The most interesting cases are in between, depending on which SQL syntactic component is variable from one execution to the other. We will briefly describe five of them.

- **Constant variability**. The only variable parts are the constant values of the query. They appear in the `where` clause of `select/update/delete` queries (as the constants with which column values, or any scalar expression value, are compared) and in `insert` queries (as the column values of the row to insert). This form has been illustrated in Query 4. A dynamic query can only appear in three successive states:

  1. *Unbound static query*, in which the variable tokens still appear as placeholders or formal variable names (see Queries 4 and 5);
  2. *Bound static query*, in which the variable tokens have been replaced with actual host variable names, providing a pure ESQL query (Query 1);
  3. *Instantiated static query*, in which the variable tokens have been replaced with constants.

- **Column variability**. The select list, that is, the list of data items in each row of the result set, may vary from one execution to the other. This form encompasses also `insert` queries, in which the value list can be variable. It generally includes constant variability as well.

- **Table variability**. The table(s) on which the statement operates may vary. All the tables must have the same structure. This form may include the first two variabilities.
- **Condition/order variability**. The selection criteria specified in the `where` clause of `select`, `update` and `delete` queries, or the `order by` clause of a `select` query, may vary.
- **Action variability**. The other parts of the SQL may vary. We can consider that the successive instantiations of the frame build quite different queries. This pattern will be found in highly interpretative applications, for instance in widespread interactive Java SQL client applications allowing users to submit any SQL statement to a database engine.

It is important to observe that a constant variability pattern can be losslessly replaced with a static SQL statement and that this property does not hold for the other patterns. For the latter, the building sequence dedicated to a given dynamic statement generates in fact a family of syntactically different static statements.

6.3 Applications of SQL statement analysis

The introduction has motivated the dynamic analysis of SQL statements as a contribution to program understanding, in particular for implicit data structure elicitation. In fact, these analysis techniques have a wider range of applications, some of which will be discussed in this section. We will develop the role of dynamic analysis in program understanding and in process control and monitoring.

6.3.1 Program understanding

SQL statement analysis contributes significantly to the more global process of program understanding. In addition, it is a major instrument to database structure understanding.

Dependency graph analysis The dependency graph of the variables of a program is a popular way to understand the coupling of different components of this program, notably for change impact analysis. This graph comprises nodes, that represent variables and constants, and edges, each of which specifies that the state of a variable depends on the state of another variable or constant. While this analysis is quite common for assignment, comparison, computing and simple input/output statements, understanding variable dependencies from rich middleware API may prove more complex. In such a context, an SQL statement can be reduced to a simple function. If only host variable dependencies are considered, Query 1 can be reduced to the following peudo-code, in which `sql1` and `sql2` are syntactic functions merely expressing uninterpreted dependencies between variables: the values of `CITY` depend, among others, on the values of `ORDID`.

```
CITY = sql1(ORDID);
PRODUCT = sql2(ORDID);
```

This transformation has been used to build the dependency graphs of program slices (Henrard et al., 1998; Cleve et al., 2006), as discussed in Chapter 5. It can be strongly improved by considering the properties of external data as expressed in the database schema. For instance, the schema of Figure 6.2 shows that (1) a functional dependency (FD) holds from column `OrdNum` to column `Sender` in table `ORDERS`, (2) an inclusion dependency holds between column `Sender` and column `CustNum` and (3) a FD holds from column `CustNum` to subcolumn `City` in table `CUSTOMER`. The transitivity rule implies that a FD also holds from column `OrdNum` to subcolumn `City` in the join `CUSTOMER*ORDER`. As a consequence, syntactical functions `sql1` and `sql2` are proved to be *mathematical* functions as well (the value of `CITY` depends on the value of `ORDID` only), so that we have proved that the result set of Query 1 will never include more than one row. This property should considerably enrich the program understanding process. As far as the we know, this approach has not been explored yet.

Implicit construct elicitation The exploitation of Query 1 illustrates the contribution of the analysis of SQL queries to the elicitation of implicit constructs and contraints of a database (Hainaut et al., 1993; Signore et al., 1994; Petit et al., 1995; Yang and Chu, 1999; Lopes et al., 1999; Shao et al., 2001; Hainaut, 2002). In addition, the enriched dependency graph that can be built by considering SQL statement interactions are used to propagate information on a node to other nodes, in particular database constructs. This is exactly the approach followed in Chapter 5 of this thesis.

Static code reconstruction Another possible application of dynamic analysis is the reconstruction of the static equivalent of dynamic SQL queries. Static analysis of static SQL statements has long been studied and analytical techniques and tools have been developed and are now available, in particular in database reverse engineering, as illustrated in Chapter 5. Rebuilding the static code can be performed at two levels of scope, namely local and global. Local SQL code analysis consider each SQL statement independently of its environment, and in particular of the other SQL statements executed before and after it. In contrast, global SQL code analysis consider also the inter-relations between successive SQL statements, mainly through shared host variables. Query 2 is a simple example of dependencies that can hold between two distant SQL statements. The latter analysis is much more powerful than the former, but it must rely on complex program understanding techniques such as program slicing (Weiser, 1984). This dynamic/static conversion can be used to reengineer complex programs in order to improve their readability and to ease their maintenability and evolvability.

Quality assessment Logging all the SQL statements issued by a program in a definite period provides a fairly comprehensive sample of all the SQL forms that the program actually uses. This information can be used to identify the SQL programming style and, in particular, *bad smells* (Mantyla, 2003) that should be reengineered.

Database usage matrix Statement analysis provides partial information that can be used to redocument the programs. A simple though quite useful derived information is the usage matrix (van Deursen and Kuipers, 1998) that specifies which tables and which columns each program unit uses and modifies. Formal concept analysis can then be applied to identify potential program clustering or database schema partitioning.

6.3.2 Process control and monitoring

Analysing the statement flow and the data flow in specific critical program points can yield precise information on the behaviour of the program at execution time. Five applications are described below.

Statistics Logging SQL statements produces a data collection that can be mined to extract aggregated information and statistics on the behaviour of the program as far as database interaction is concerned. According to the additional data recorded with these statements, useful information can be derived such as database failure rate at each program point (update rejected, access denied, empty result set, etc.), most frequently used SQL forms, complexity of SQL statements, programming idiosyncrasies, awkward and inefficient SQL patterns and non-standard syntax (inducing portability vulnerability). These derived data can be used to monitor the program behaviour but also for refactoring purpose, for instance to improve the quality and the portability of the source code. The analysis of selection criteria in select, delete and update statements can be used to define and tune the data access mechanisms such as indexes and clusters.

Accounting The execution of a program can be charged information access cost. This cost can depend on the value of the information, on the volume of the data extracted from the database or on the time the database engine spent on executing the queries.

Performance analysis Time recording before and after each SQL statement execution allows a precise evaluation of that part of program execution spent on data exchange with the database. Analysing these data can be used to fine tune both the application program and the database for better performance.

Transaction management The DBMS takes in charge transaction management according to the policy defined by the database administrator. Ad hoc, customized, transaction management can be necessary in some critical situation. Capturing all the data modification SQL statements submitted to the DBMS at run time and recording them in a log make it possible to redo (and in some circumstances to undo) their effect when some adverse events or conditions occur.

Security The main database application vulnerability is *SQL code injection* (Halfond and Orso, 2005; Merlo et al., 2006; Lam et al., 2008). It occurs when an external user is requested to provide data (typically its user ID and password) that are injected at building time into an incomplete SQL query. However, the user actually provides, instead of the expected data, spurious but syntactically valid data in such a way that a malware query is formed. This query is then executed with privileges that the user has not been granted. For instance, a user who logs in a system is requested to provide its password through a dialog box. When aquired, the value is injected into an SQL query to form a valid statement that checks the existence of this password in the authorization table. The instantiated statement generally looks like the following: `"select count(*) from USERS where PASSWORD = 'x1123bz';"`, and is expected to return a zero (access denied) or non-zero (access granted) value. It has been built by injecting the value `x1123bz` provided by the user according to the frame `"select count(*) from USERS where PASSWORD = '" + "x1123bz" + "';"`. If the user enters the string `"X' or 'A' = 'A"` instead of a valid password, the statement built becomes the unexpected but valid statement `"select count(*) from USERS where PASSWORD = 'X' or 'A' = 'A';"`. Since this query returns the number of rows in the table, the unauthorized user is given access to the system. Interestingly, the problem can be formalised as the illegal transformation of the constant variability pattern written by the programmer into an unexpected condition variability pattern. Most common attack detection techniques rely on the analysis of the value provided by the user, but dynamic analysis of the actual (vs intended) SQL query may prove easier and more reliable. In the same domain, SQL code analysis can detect unauthorized queries and updates. For instance, a query including the fragment `"from REPORT where STATUS = 'classified'"` can be identified as suspect and blocked until official authorization notification. The analysis of the result set of SQL queries can also be used to identify the presence of sensitive information.

6.4 SQL statement capturing techniques

At runtime, the SQL protocol relies on a dataflow that starts and ends at the host program point defined by an SQL statement. The successive steps define the flow points at which capture instruments can be installed. The following eleven steps are identified.

- Step d1. (Building step, client side) The SQL statement is formed by the concatenation of statement fragments. The resulting string generally includes constant placeholders.

- Step d2. (Preparation step, client side) The statement is sent to the database engine for preparation.
- Step d3. (Binding step, client side) The placeholders, if any, are bound with host variables so that the statement is now complete.
- Step d4. (Transmission step, client side) The host program transmits the SQL statement to the client API for execution. The program passes control to the API and suspends itself.
- Step d5. (Statement sending step, link side) The client API receives the SQL statement and sends it to the database engine.
- Step d6. (Statement receiving step, link side) The database engine receives the statement and writes it on its log.
- Step e1. (Recompilation step, server side) The database engine checks the validity of execution plan of the query by comparing the compilation date of the query with the last update time of the schema.
- Step e2. (Execution step, server side) The database engine executes the query.
- Step b1. (Result step, server side) The database engine sends the status messages to the client and, if needed, the result set is extracted from the database.
- Step b2. (Receiving step, link side) The client API receives the message and result set.
- Step b3. (Extraction step, client side) The client receives control from the API and extracts the results to store them in its host variables.

It is important to note that each API variant executes a subset only of these steps. For instance, in static SQL, steps d1, d2, d3 are performed at writing time and compile time; similarly, b3 is a compile time step. In dynamic SQL, step e1 is absent.

Below, we describe ten techniques to capture the SQL statements that are executed in a data-intensive application program. Among them, seven techniques (summarized in Figure 6.3) are intended to understand the behaviour of the client/server system at execution time. The three other techniques rely on static analysis.

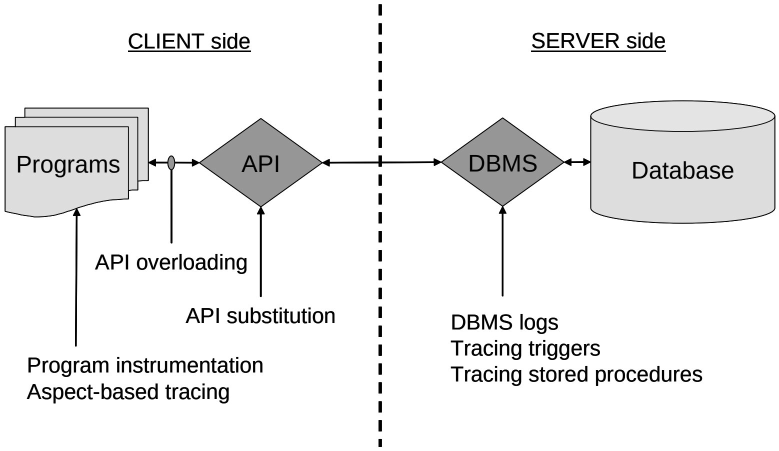


Figure 6.3: Seven capturing techniques for SQL statement executions.

```
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(SQLstmt);
SQLlog.write(”132;SQLexec;”+stmt.hashCode()+”;”+SQLstmt);
rs.next();
vName = rs.getString(1);
SQLlog.write(”133;SQLgetS1;”+rs.getStatement().hashCode()+”;”+vName);
vSalary = rs.getInt(2);
SQLlog.write(”134;SQLgetI2;”+rs.getStatement().hashCode()+”;”+vSalary);
```

Figure 6.4: Logging SQL operations by program instrumentation.

6.4.1 Program instrumentation

The capture of a dynamic SQL statement is performed by a dedicated code section inserted before the program point of this statement. Similarly, the result of an SQL statement will be captured by a code section inserted after it. This technique requires code analysis to identify and decode database API statements and entails source code modification and recompilation. It provides a temporal list of statement instances. In the example of Figure 6.4, the tracing statement writes in the log file the program point id (132), the event type (SQLexec), the statement object id (`stmt.hashCode()` or `rs.getStatement().hashCode()`) followed by the SQL statement (building time constant injection is assumed) or the output variable contents. According to the information that needs to be extracted from the trace, program id, process id and/or timestamp can be output as well. This technique may involve more complex source code restructuring when the SQL statement is built in the SQL prepareStatement argument (as in prepareStatement(A+B+C)). Indeed, this building can invoke functions with side effects such as acquiring statement fragments from external sequential sources. It should be noted that static SQL can be processed in the same way. Since the statement explicitly appears in

```
package myAPI;
...
public class Statement{
  java.sql.Statement stat;

  public Statement(java.sql.Connection con){
  stat = con.createStatement();
  }
  ...
  public ResultSet executeQuery(String sql){
       log.traceQuery(sql);
       return new ResultSet(stat.executeQuery(sql));
  }
  ...
}
```

Figure 6.5: Illustration of API overloading.

the source code, only the values of input and output variables need to be captured. The advantage of dynamic analysis of static SQL is that it provides information on statement instances. However, it does not yield any information on statements that are not executed.

6.4.2 Aspect-based tracing

Aspect technology, if available for the programming language of interest, allows triggers to be added to an existing program without source code modification. The introduction of tracing aspects simply requires the programs to be recompiled. In the case of Java/JDBC applications, AspectJ pointcuts and advices can be defined in order to capture the successive events involved in dynamic database manipulation like query preparation, input value definition, query execution and result extration (Cleve and Hainaut, 2008). Aspect-oriented support is now available for several programming languages among which Java, C, C++, ... and COBOL (Lämmel and De Schutter, 2005).

6.4.3 API overloading

The API overloading technique consists in encapsulating (part of) the client side API within dedicated classes which provide similar public methods but produce, in addition, the required SQL execution trace. For instance, we could write our own `Statement`, `PreparedStatement` and `ResultSet` classes which, in turn, make use of the JDBC corresponding classes, as shown in Figure 6.5. This technique requires a minimal program adaptation (illustrated in Figure 6.6) and, consequently, recompilation.

6.4.4 API substitution

If the source code of the client side API is available, which is the case for ODBC and JDBC drivers of Open Source RDBMSs, tracing statements can be inserted directly

```
import java.sql.PreparedStatement;         import myAPI.PreparedStatement;
import java.sql.ResultSet;                 import myAPI.ResultSet;
import java.sql.Statement;                 import myAPI.Statement;
...                                    --> ...
Statement sta = con.createStatement();     Statement sta = new Statement(con);
ResultSet rsl = sta.executeQuery(q);       ResultSet rsl = sta.executeQuery(q);
```

Figure 6.6: Program adaptation for API overloading.

in this API. The latter is then recompiled and bound to the client applications. The client program need not be modified nor recompiled. This technique records the statement instances but ignores the program points.

6.4.5 DBMS logs

Most database engines store the requests received from the client application programs in a specific file or table. For example, MySQL writes, in the order it received them, all the client queries in its general query log. Each record comprises the client process id, the time stamp the query is received and the text of the query as it was executed, in particularly with input variables replaced with their values. As compared to the trace obtained by the insertion of tracing statement technique, DBMS log does not provide program points and can be processed off-line only. This technique does not require source code modification.

6.4.6 Tracing triggers

A tracing trigger is an SQL trigger created to capture and record data update activities on a definite table. The body of the trigger is executed before or after each data modification query. It is not provided with the original text of the query but its effect can be recovered by the comparison of the states of the data before and after query execution. By combining for-each-statement and for-each-row triggers, elementary fictitious queries can be reconstructed and recorded for further analysis. This technique is weaker than most others since it cannot capture the actual statements and since it ignores consultation queries, but it implies no source code modification. For instance, let us consider an update trigger executed for each update query instance. An equivalent fictitious query can be built as follows: for each column `C` for which `new.C <> old.C`, a set clause `"set C=" + new.C` is defined. A trigger monitoring update queries with invariant primary key for the table `ORDERS` of Figure 6.1 would be defined by the pseudocode shown in Figure 6.7.

6.4.7 Tracing stored procedures

Another possible technique consists in replacing some or all SQL statements in client programs by the invocation of equivalent SQL procedures created in the database. The set of these procedures acts as an ad hoc API that can be augmented

```
create trigger LOG_ORDER_UPDATE before update on ORDERS
for each row
begin
 declare S varchar(500);
 S := "update ORDERS ";
 if new.ORDDATE <> old.ORDDATE
    then S := S + "set ORDDATE=" + new.ORDDATE;
 if new.REFERENCE <> old.REFERENCE
    then S:= S + ", set REFERENCE=" + new.REFERENCE;
 if new.SENDER <> old.SENDER
    then S:= S + ", set SENDER=" + new.SENDER;
 S = S + " where ORDNUM=" + old.ORDNUM + ";";
end;
```

Figure 6.7: Reconstructing a fictitious update query using an update trigger

with tracing instructions that maintain a log of the instances of SQL statements that are executed. This technique can be considered in architectures that already rely on SQL procedures. When client programs include explicit SQL statements, it entails in-depth and complex code modification. However, since it replaces complex input and output variable binding with mere procedure arguments, this reengineering can provide a better code that will be easier to maintain and evolve.

6.4.8 Static analysis of dynamic SQL statements

When SQL statements are built according to a disciplined and systematic procedure, static analysis techniques such as mere sequential parsing or program slicing can be used to understand this procedure and reconstruct the static query. This technique allows to retrieve the source code locations of the queries, but it cannot trace query *instances* (i.e., queries where each variable/placeholder is replaced with a value). The example of Query 3 provides a clear illustration of such a scenario, that has been studied by van den Brink et al. (2007) in the context of quality assessment.

6.4.9 Static analysis of static SQL statements

As Query 1 shows, the identification of static SQL statements is immediate, all the more since these statements are signalled by specific meta-instructions such as `exec SQL` and `end-exec` in COBOL, `#SQL{...}` in SQLJ or equivalent in other languages. Their further analysis poses no hard problems since the statements are guaranteed to be correct. This technique provides statement program points but does not trace their instances. Global static analysis consists in identifying the dependency relationships between SQL statements. Query 2 illustrates the point by explicitly showing that output variable `CUST` of the first query is used as input variable in the second one. However, this identification may prove much more complex than local analysis when there is no simple and transparent path from the output variables of a statement and the input variable of a subsequent statement (Willmor et al., 2004).

6.4.10 Extraction of compiled queries from system tables

Programs relying on static SQL must be precompiled. Through this process, each SQL statement is analysed by a precompiler that stores both the source statement and its compiled equivalent in a DBMS system table (step e1 above). Generally, this table can be read off-line, so that all the SQL statements of a program can be examined. This procedure does not provide statement instances, host variable values nor program points. It does not apply to dynamic SQL statements, that are compiled for each execution of the client program.

6.5 Evaluation and applicability of SQL capturing techniques

In this section, we will evaluate and compare the SQL capturing techniques identified in Section 6.4. To this aim, they will be evaluated and compared against functional and operational criteria that we will study first. The evaluation criteria will be classified into four property families, namely the nature of the information captured, completeness, host languages and operational characteristics.

Nature of information captured. Each technique captures information that will be further processed. This information can be the static SQL statement, which comes in three variants, namely unbound (with variable placeholders), bound (with actual host variable names) and instantiated (with constants). The technique can also return the result of the execution of the statement as data values and/or execution status. The program point of the statement can be returned as well. It appears that some techniques cannot cope with all the SQL, the class of statement that can be captured will be specified: DDL (`create`, `drop`, `alter`), data extraction (`select`), data modification (`insert`, `update` and `delete`), control (`grant`, `revoke`, etc.) and binding statements that connect host variables to the unbound input and output placeholders.

Completeness. The question is, can all the SQL statements of the client program be captured by the technique. Due to its complexity in the context of dynamic SQL, this question will only be mentioned for static SQL statements. We will distinguish the identification of statements and that of statement instances. Ngo and Tan (2008) define an automatic procedure to identify as many as possible SQL statement instances (they call them concrete statements) through static analysis techniques based, notably on symbolic execution of program paths that include database interactions. Actual case studies show that about 80% of instantiated statements can be identified. None of the techniques described in this chapter address this problem, so that we will discuss the completeness characteristic for static statements only.

Host languages. A technique may be applicable to some host languages and not to others. We will examine whether each technique is applicable to COBOL, C, C++ and Java.

Operational characteristics. Some techniques impose more or less strong requirements on the following aspects:

- *Database schema*: does the technique require schema *examination*, schema *modification*?
- *Client source code*: does the technique require code *examination*, code *modification*, code *recompiling*?
- *Cost*: each technique induces various kinds of additional cost. We distinguish preparation cost, client runtime cost and server runtime cost. In each category, a coarse-grain score of 0 (no to low cost), 1 (medium cost) and 2 (high cost) will be assigned.
- *Processing time*: the information captured allow real-time processing (RT) or requires deferred processing (D). This property is crucial for such applications as security control.

Table 6.1 synthetizes the characteristics of each SQL statements capturing technique according to these properties.

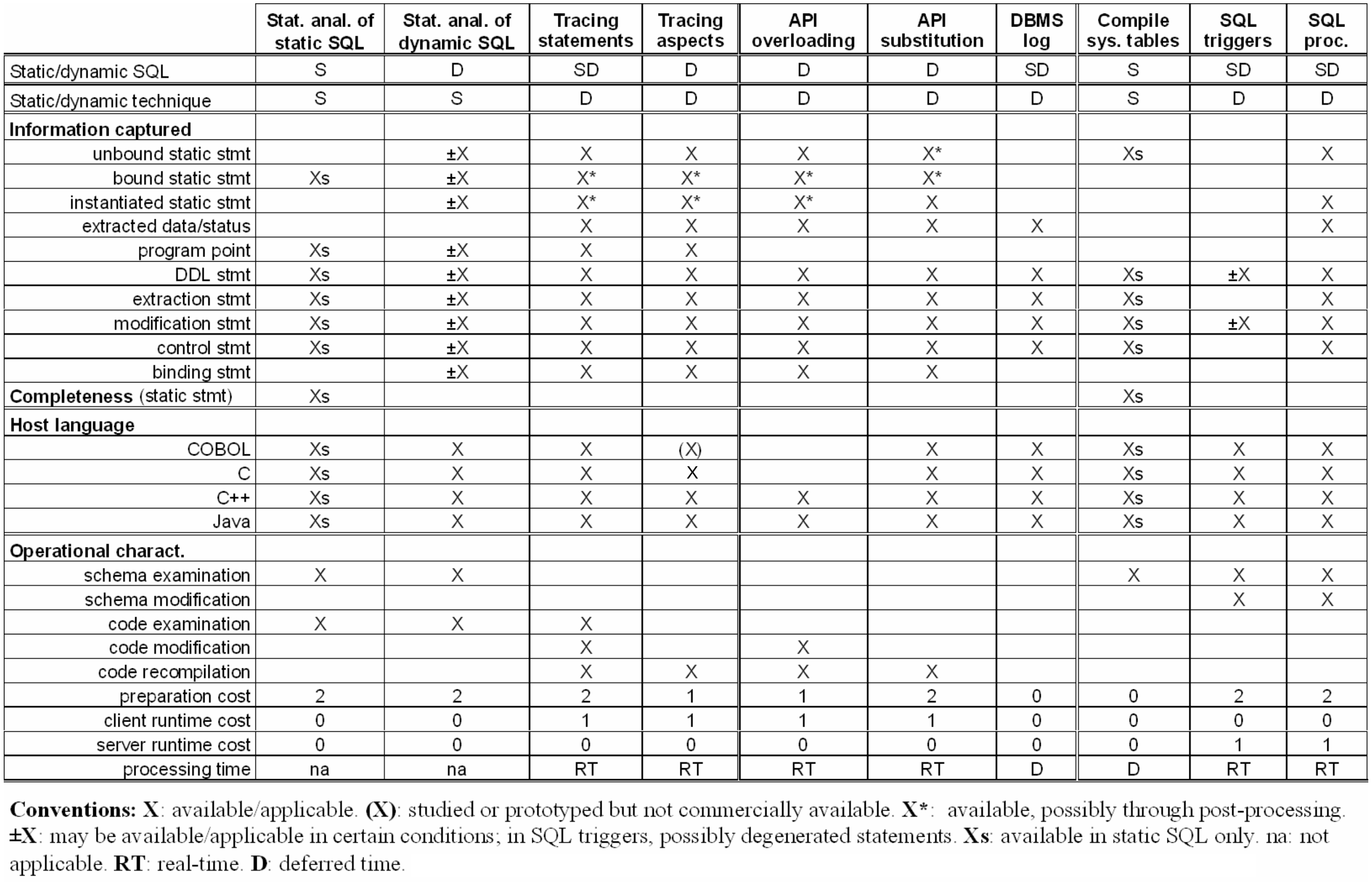
6.6 Aspect-based dynamic analysis

In this section, we will particularly elaborate on the use of tracing aspects for capturing SQL execution traces (Cleve and Hainaut, 2008). This technique has many advantages. First, it does not require any source code modification, but a recompilation only. Second, an aspect may retrieve corresponding source code locations when capturing an execution event. Finally, it allows iterative and incremental analysis to be performed with minimal effort.

Aspect-based tracing consists in specifying separately the tracing functionality by means of *aspects*, without any alteration of the original source code. An aspect typically consists of *pointcuts* and associated *advices*, which can be seen as program-side *triggers*. A pointcut picks out certain *join points* in the program flow, which are well-defined moments in the execution of a program, like method call, method execution or object instantiation. An advice is associated to a pointcut. It declares that certain code should execute at each of the join points specified by the pointcut. Depending of the kind of advice, the code is run *before*, *after*, or *around* the specified join point.

In this chapter, the tracing aspects are written in AspectJ (Kiczales et al., 2001), an aspect-oriented extension to Java. The pointcuts used refer to method call join points that correspond to SQL statement construction and execution.

Table 6.1: The SQL statements capturing techniques and their characteristics.



	Stat. anal. of static SQL	Stat. anal. of dynamic SQL	Tracing statements	Tracing aspects	API overloading	API substitution	DBMS log	Compile sys. tables	SQL triggers	SQL proc.
Static/dynamic SQL	S	D	SD	D	D	D	SD	S	SD	SD
Static/dynamic technique	S	S	D	D	D	D	D	S	D	D
Information captured										
unbound static stmt		±X	X	X	X	X*		Xs		X
bound static stmt	Xs	±X	X*	X*	X*	X*				
instantiated static stmt		±X	X*	X*	X*	X				X
extracted data/status			X	X	X	X	X			X
program point	Xs	±X	X	X						
DDL stmt	Xs	±X	X	X	X	X	X	Xs	±X	X
extraction stmt	Xs	±X	X	X	X	X	X	Xs		X
modification stmt	Xs	±X	X	X	X	X	X	Xs	±X	X
control stmt	Xs	±X	X	X	X	X	X	Xs		X
binding stmt		±X	X	X	X	X				
Completeness (static stmt)	Xs							Xs		
Host language										
COBOL	Xs	X	X	(X)		X	X	Xs	X	X
C	Xs	X	X	X		X	X	Xs	X	X
C++	Xs	X	X	X	X	X	X	Xs	X	X
Java	Xs	X	X	X	X	X	X	Xs	X	X
Operational charact.										
schema examination	X	X						X	X	X
schema modification									X	X
code examination	X	X	X							
code modification			X		X					
code recompilation			X	X	X	X				
preparation cost	2	2	2	1	1	2	0	0	2	2
client runtime cost	0	0	1	1	1	1	0	0	0	0
server runtime cost	0	0	0	0	0	0	0	0	1	1
processing time	na	na	RT	RT	RT	RT	D	D	RT	RT

Conventions: **X**: available/applicable. **(X)**: studied or prototyped but not commercially available. **X***: available, possibly through post-processing. **±X**: may be available/applicable in certain conditions; in SQL triggers, possibly degenerated statements. **Xs**: available in static SQL only. na: not applicable. **RT**: real-time. **D**: deferred time.

```
 1 public aspect SQLTracing {
 2
 3  private MySQLLog log = new MySQLLog();
 4
 5  pointcut queryExecution(String query):
 6    call(ResultSet Statement.executeQuery(String)) && args(query);
 7
 8  before(String query): queryExecution(query){
 9    String file = thisJoinPoint.getSourceLocation().getFileName();
10    int LoC = thisJoinPoint.getSourceLocation().getLine();
11    Statement statement = (Statement) thisJoinPoint.getTarget();
12    log.traceQuery(file, LoC, statement.hashCode(), query);
13  }
14 }
```

Figure 6.8: Tracing SQL query executions

6.6.1 Capturing query executions

Figure 6.8 shows a simple tracing aspect that captures and records the execution of an SQL query (without statement preparation). The declared pointcut (lines 5-6) refers to the invocation of method `executeQuery` of class `Statement`. Before each occurence of query execution, the advice (lines 8-13) writes a log entry indicating (1) the class name, (2) the line of code, (3) the object id of the statement and (4) the query string itself.

The way of tracing SQL query executions is a bit more complicated in the presence of statement preparation. In order to reconstruct the full query string to be traced, the aspect code must be able to maintain the link between three successive events: *statement preparation*, *value injection* and *query execution*. This can be done using the technique illustrated in Figure 6.9, and that we detail below.

Statement preparation At statement preparation time, an *around* advice (lines 9-18) captures the created statement as well as the incomplete query string passed as an argument. This information is stored in a correspondence table, that maps each statement to an instance of class `PreparedStatementInfo`. The latter contains (1) the initial query string (2) the values corresponding to its placeholders (these values are still undefined at preparation time, they will be initialized at value injection time).

Value injection Another pointcut/advice couple (lines 20-29) is defined for logging the injection of query input values, performed through the invocation of a *set* method on an object of class `PreparedStatement`. The advice updates the information corresponding to the statement in which the value is injected, so that placeholder at position `pos` is now associated to input value `val`.

```
public aspect SQLTracingWithStatementPreparation{

 private MySQLLog log = new MySQLLog();
 private MyStatementTable statementTable = new MyStatementTable();

 pointcut statementPreparation(String query) :
   call (PreparedStatement Connection.prepareStatement(String))
   && args(query);

 Object around(String query) : statementPreparation(query){
   String file= thisJoinPoint.getSourceLocation().getFileName();
   int LoC = thisJoinPoint.getSourceLocation().getLine();
   Object stat = proceed(query);
   PreparedStatementInfo info = new PreparedStatementInfo(query);
   statementTable.store(stat, info);
   log.traceStatementPreparation(file, LoC, stat.hashCode(), query);
   return stat;
 }

 pointcut valueInjection(PreparedStatement stat, int pos, Object val) :
   call (void PreparedStatement.set*(int, *))
   && target(stat) && args(pos, val);

 before(PreparedStatement stat, int pos, Object val) : valueInjection(stat,
 pos, val){
   String file = thisJoinPoint.getSourceLocation().getFileName();
   int LoC = thisJoinPoint.getSourceLocation().getLine();
   String methodName = thisJoinPoint.getSignature().getName();
   statementTable.lookup(stat).set(pos, val);
   log.traceValueInjection(file, LoC, stat.hashCode(), methodName, pos, val);
 }

 pointcut queryExecution(PreparedStatement stat) :
   call(ResultSet PreparedStatement.executeQuery()) && target(stat);

 before(PreparedStatement stat) : queryExecution(stat) {
   String file= thisJoinPoint.getSourceLocation().getFileName();
   int LoC = thisJoinPoint.getSourceLocation().getLine();
   PreparedStatementInfo info = statementTable.lookup(stat);
   String fullQuery = info.getFullQuery();
   log.traceQuery(file, LoC, stat.hashCode(), fullQuery);
 }
}
```

Figure 6.9: Tracing SQL query executions in the presence of statement preparation

```
pointcut resultExtraction(ResultSet rSet) :
  call(** ResultSet.get*(**)) && target(rSet);
Object around(ResultSet rSet) throws SQLException : resultExtraction(rSet){
  String file= thisJoinPoint.getSourceLocation().getFileName();
  int LoC = thisJoinPoint.getSourceLocation().getLine();
  String methodName = thisJoinPoint.getSignature().getName();
  Object colNameOrInd = thisJoinPoint.getArgs()[0];
  Object res = proceed(rSet);
  Statement stat = rSet.getStatement();
  log.traceResult(file, LoC, stat.hashCode(), methodName, colNameOrInd, res);
  return res;
 }
```

Figure 6.10: Tracing SQL result extraction

Query execution The query execution advice (lines 34-40) has almost the same behaviour as the one of Figure 6.8, except that the full query string first has to be produced from the statement information stored in the correspondence table.

6.6.2 Capturing query results

Dynamic analysis can also be used to examine the results of SQL code execution. The format of the output data is defined by the form of the SQL query and the information of the logical schema. Result analysis often is the only technique to trace the link between two SQL queries through shared host variables when the intermediate host statements cannot be analysed statically.

Figure 6.10 gives an example of aspect code capturing SQL query results, whatever the statement class used (`Statement` or `PreparedStatement`). The pointcut is associated to a *get* method on an instance of class `ResultSet`. The advice logs (1) the source code location, (2) the statement identifier (3) the name of the *get* method invoked, (4) the name or index of the corresponding column and (5) the result value itself.

6.6.3 Detecting query imbrication

Tracing aspects such as those presented above may also be used to dynamically detect potential dependencies between successive query executions. A possible technique consists in analyzing the imbrication relationship between the SQL statements. A query q_2 is said to be *imbricated* if its execution is performed *before* the result set of the preceding query q_1 has been completely emptied. Such a situation strongly suggests that a data dependency holds between the output values of q_1 and the input values of q_2.

Detecting imbricated queries can be performed using the following technique. The aspect makes use of two status fields while tracing SQL executions. The first field represents the query imbrication level, and the other one consists of a FIFO pile of recorded queries. Each time a query is executed, the imbrication level is incremented, and the query (together with its source code location) is put on top of the pile. Conversely, each time a `ResultSet` object is emptied (i.e., when a `ResultSet.next()` invocation returns false), the aspect decrements the imbrication level, and removes the top element from the pile. Thus, if the imbrication level is greater than zero at the time of executing a new query q, it means that q is imbricated. We can then make the hypothesis that there exists a dependency between the query being at the top of the current pile, and query q. Unfortunately, this technique fails in case the program does not use the complete result set of (some of) the queries it executes, which can be considered as unfrequent.

6.6.4 Analyzing input and output values

Another way to detect inter-dependent successive queries is to analyse the link between their respective input and output values. This can be done easily in the presence of statement preparation. The tracing aspect can store the values extracted from a resultset in order to compare them to the values injected as inputs of the following queries. The same analysis can be performed by comparing input values of a query with input values of another subsequent query.

6.7 SQL trace processing

Once one or several SQL traces have been produced from program executions, they need to be further analyzed and interpreted. The complexity of the process obviously depends on its objective. Applications of dynamic analysis of dynamic SQL statements rely on lower-level, intermediate objectives that can be classified in two major categories, namely identification of the varying components in constant variability patterns and the reconstruction of static statements equivalent to dynamic patterns.

As discussed by Ngo and Tan (2008), identifying all the variants of static statements that can be generated by the dynamic SQL statements of a program still is an open problem. In the framework developed in this chapter, this means that, in general, as long as we are not able to generate the set of instantiated static statements that could be generated at a given program point, we cannot determine the category of variability patterns a dynamic statement generation code section belongs to. The following analysis considers trace examination and mining only. They could be refined by taking into account the contribution of (partial) static analysis. For instance, the category of variability can sometimes be determined by static analysis (van den Brink et al., 2007; Ngo and Tan, 2008), as illustrated by Query 3.

6.7.1 Constant/variable identification

The objective is to identify, in instantiated statements, the constants that may vary depending on the execution and to establish the link between these constants and the host variables. The trace of any SQL statement that is, or has been, submitted for execution is an instantiated static statement.

Variable constants in constant variability pattern. If a significantly large set of traces produced at the same program point vary on the constants only, they can be considered as generated by a constant variability pattern. A problem arises for *slowly varying* constants. This phenomenon concerns query parameters related to spatio-temporal aspects of program execution. If a query includes a condition on the current year or on the branch of the company, dynamic analysis of the trace generated from a given workstation during a given month cannot identify such constants as possibly varying. However, if the trace also includes variable binding statements, as in Query 5, matching the placeholder or formal variable and the binding couple allows variability to be explicitly identified.

Input and output host variables of a constant variability pattern. Once the constants have been identified as varying, it can be necessary to identify their data sources in the program code, that is, in most case, the host input variables. When static analysis fails, as in Query 7, data analysis can detect constant equality in the traces from different program points. If the first constant belongs to the result of query q_1, if the second constant is an input value of query q_2 and if both constants are equal throughout the traces, then the probability that both constants come from the same variable is high. Studying input and output constants may also reveal another kind of inter-query dependency, according to which the result of a query q_1 influences the fact that another query q_2 is executed or not. This is typically the case when a program first verifies an implicit integrity constraint (as a foreign key) before inserting or updating an SQL row.

6.7.2 Static statements reconstruction

Several major applications of SQL analysis require the reconstruction of the static statement(s) generated by a definite dynamic pattern. Therefore, techniques for deriving these statements from the trace are essential when static analysis cannot completely identify them from the source code.

- *Instantiated statements in constant variability pattern.* As long as the executed statements are traced, all instantiated statements are included in the trace. All is needed is to check that there is no other variability than constant variability.

- *Bound statements in constant variability pattern.* Replacing constants by their data sources yields the exact static equivalent of a dynamic SQL statement. This process, that leads to the identification of the program constructs that supply the constants, is based on both static analysis and trace processing. It can be particularly complex when a constant is built by an expression *in situ* (`CustNum = 'C' + :CODE`) instead of merely extracted from a variable (`CustNum = :CNUM`). Introducing an additional variable in the host code will solve the problem.

```
12   query = "select Address from CUSTOMER where Num = ?";
13   SQLstmt = connection.prepareStatement(query);
14   SQLstmt.setInt(1, vCNUM);
15   rset = SQLstmt.executeQuery();
16   rset.next();
17   vADDRESS = rset.getString(1);
```

```
  File         Line#  Statement# Details
  -------------------------------------------------------------------------------
  AClass.java;  13;    123456;     select Address from CUSTOMER where Num = ?
  AClass.java;  14;    123456;     setString(1, ’C400’)
  AClass.java;  15;    123456;     executeQuery()
  AClass.java;  17;    123456;     getString(1) = ’10, Downing Street, London’
```

Figure 6.11: A JDBC code fragment together with a corresponding execution trace

- *Unbound statements in constant variability pattern.* Though less useful in program understanding, it can be used to standardize programming style. Aligning all dynamic patterns of a program on Query 5 or Query 6 styles is an example. If the unbound statements have not been traced, then they can easily be derived from their bound versions.

- *Family of constant variability patterns.* When statement parts other than the constants are varying, then the dynamic pattern generates a family of static statements instead of a single statement. Considering a definite execution (or preparation) program point, the syntactic analysis of its instantiated (or bound/unbound) static statement produces a set of constant variability patterns. If each pattern is induced from a significantly large subset of traces, then each pattern can be considered, with a high probability, as a pertinent static statement. As already mentioned, the completeness problem of the set of constant variability patterns is still unsolved.

Illustration

SQLJ was designed to compensate for the poor readability of JDBC program patterns. For example, the dynamic code section of Query 5 could be rewritten as the following static SQLJ statement:

```
#sql{Select OrdNum into :Num from ORDERS where Sender = :CNum}
```

Statement preparation, input variable binding, execution and result extraction statements are merged into a single statement that explicitly shows the query architecture as well as the input and output host variables. The resulting program is much more readable and far easier to maintain.

Figure 6.11 shows a JDBC code fragment together with a corresponding execution trace obtained with a tracing aspect. Each entry of the trace contains the object id of the `Statement` involved in the query execution (such an id is obtained via `SQLstmt.hashCode()`). This allows the correct recovery of all the steps involved in a given query execution at a specific program point. By combining and exploiting the information provided by each inter-related event, both the query instance and, as a second step, the static query can be reconstructed. We illustrate this process by analyzing the trace of Figure 6.11. We first notice that a query was executed at line 15. Based on the statement number, we can go backward in the trace to understand the way the executed query was constructed. The statement preparation trace entry provides us with a query string including an input value placeholder, while the value injection entry reveals the source code location where this placeholder was replaced with an actual input value using method `setInt`. At this point, we are able to produce the query instance which was actually executed at line 15:

```
select CustAddress from CUSTOMER where Num = 'C400'
```

Concerning the static form of the query, we can derive from the trace that method `executeQuery()` of line 15 actually executes an SQL statement of the form `select CustAddress from CUSTOMER where Num =` v, where v is the variable/constant used as second argument of method `setInt` of line 14. This line shows that v actually is host variable (`vCNUM`). Similarly, the `into` clause of the static query can be obtained by retrieving the variable to which the result of method `getString(1)` of line 17 is assigned (`vADDRESS`). We now obtain the following complete static SQLJ query:

```
#sql {select Address into :vADDRESS from CUSTOMER where Num = :vCNUM}
```

6.8 Application to database reverse engineering

As illustrated in the introduction, analyzing SQL queries can help eliciting implicit database schema constructs and constraints among which undeclared foreign keys, identifiers and functional dependencies (Petit et al., 1995; Lopes et al., 1999; Tan et al., 2002; Tan and Zhao, 2003). In this section, we will illustrate the use of SQL execution trace analysis as a basis for formulating hypotheses on the existence of an undeclared foreign key. These hypotheses will still need to be validated afterwards (e.g., via data analysis or user/programmer interviews).

We can distinguish two different approaches to detecting implicit referential constraints between columns of distinct tables. We can either observe the way such referential constraints are *used* or the way they are *managed*.

- *Referential constraint usage* consists in *exploiting* the referential constraint. For instance, within the same execution, an output value o_1 of an SQL statement s_1 querying table T_1 is used as an input value of another SQL statement s_2 accessing another table T_2. A more direct usage of a foreign key consists in a join of T_1 and T_2 within a single query. In both cases, this could suggest the existence of an implicit foreign key between tables T_1 and T_2.

- *Referential constraint management* aims at *verifying* that the referential constraint keeps being respected when updating the database. For instance, before modifying the content of a table T_2 (by an `insert` or `update` statement s_2), the program executes a verification query q_1 on table T_1. According to the result of q_1, s_2 is executed or not. When both q_1 and s_2 are executed, they contain at least one common input value. Similarly, when deleting a row of a table $T1$ using a `delete` statement d_2, the program also deletes a possibly empty set of rows of another table T_2 via a another delete statement d_1 (procedural delete cascade).

In summary, we can identify three main heuristics for implicit foreign key constraints detection from SQL execution traces, namely *join*, *output-input dependency* and *input-input dependency*.

6.8.1 Joins

As already suggested above, SQL joins often rely on the matching of a foreign key and a primary key. The join of Query 1 corresponds to an standard join, where several tables occur in the `from` clause of the query. It combines the rows of those tables, typically based on a given *join condition*. The SQL language also provides the programmer with *explicit* join operators for building the join of two or more tables.

There exist two main categories of SQL joins, namely *inner joins* and *outer joins*.

Inner joins An inner join represents the default join type. The outcome of an inner join can be obtained by first taking the *cartesian product* of all rows of the tables, and then selecting all the resulting rows which satisfy the join condition. We can distinguish several sub-categories of inner joins, including:

- *Cross-join*, where the join condition is absent (or always evaluated to true). For instance, one can write:

  ```
  select *
  from CUSTOMER cross join ORDERS
  where ...
  ```

  which is equivalent to the following implicit join:

  ```
  select *
  from CUSTOMER, ORDERS
  where ...
  ```

- *Equi-join*, where the join condition is based on the *equality* of columns from the joined tables, as in the following query:

  ```
  select *
  from CUSTOMER C join ORDERS  O
  on (C.ncus = O.ncus)
  where ...
  ```

The above query can also be expressed as follows:

```
select *
from CUSTOMER C, ORDERS  O
where C.ncus = O.ncus
and ...
```

- *Natural join*, which is a special kind of equi-join where the join condition is *implicit*, i.e., relying on the comparison of all the columns that have the same column name in the joined tables. An example of a natural join is given below:

```
select *
from CUSTOMER natural join ORDERS
where ...
```

Outer joins An outer join, as opposed to an inner join, does not require each row in the joined tables to have a matching row. The joined table retains each row, even if no other matching row exists. Outer joins subdivide further into *left outer joins*, *right outer joins*, and *full outer joins*, depending on which table(s) one retains the *orphan* rows from. The left (resp. right) outer joins retain the orphan rows form the table occuring at the left (resp. right) of the join operator. The full outer join preserves the orphan rows of all the joined tables in the result.

In the context of implicit foreign key discovery, equi-joins are of particular interest. Indeed, join conditions expressing the equality between columns of several tables may be seen as an indication of a referential constraint usage.

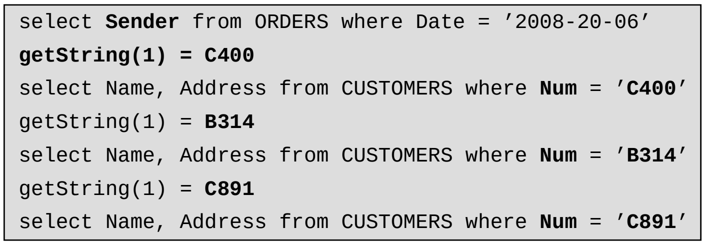
6.8.2 Output-input dependency

Notations Let q be a SQL query. Let $q.in$ be the set of input values of q, and $q.out$ be the set of output values of q. Let $q.seq$ be the sequence number of query q in the trace.

Definition 1 *A query q_2 is* output-input dependent *on another query q_1 iff*

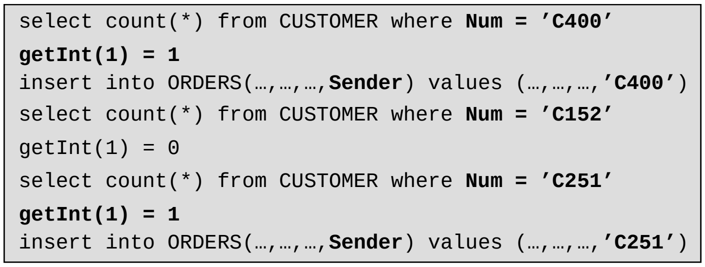
- $q_2.in \cap q.out \neq \emptyset$
- $q_2.seq > q_1.seq.$

In the case of a *procedural* join between the source and target tables of an implicit foreign key, the value of the foreign key column(s) may be used to retrieve the target row using a subsequent query, as shown in Figure 6.12. Conversely, the value of the identifier of a given target row can be used to extract all the rows referencing it.



```
select Sender from ORDERS where Date = ’2008-20-06’
getString(1) = C400
select Name, Address from CUSTOMERS where Num = ’C400’
getString(1) = B314
select Name, Address from CUSTOMERS where Num = ’B314’
getString(1) = C891
select Name, Address from CUSTOMERS where Num = ’C891’
```

Figure 6.12: An execution trace with output-input dependencies that may reveal the existence of the implicit foreign key of Figure 6.1



```
select count(*) from CUSTOMER where Num = ’C400’
getInt(1) = 1
insert into ORDERS(…,…,…,Sender) values (…,…,…,’C400’)
select count(*) from CUSTOMER where Num = ’C152’
getInt(1) = 0
select count(*) from CUSTOMER where Num = ’C251’
getInt(1) = 1
insert into ORDERS(…,…,…,Sender) values (…,…,…,’C251’)
```

Figure 6.13: An execution trace with input-input dependencies that may reveal the existence of the implicit foreign of Figure 6.1

6.8.3 Input-input dependency

Definition 2 *A query* q_1 *is* input-input dependent *on another query* q_2 *iff* $q_1.in \cap q_2.in \neq \emptyset$.

As an example, let us consider the execution trace given in Figure 6.13. This trace strongly suggests the existence of an implicit foreign key between column `Sender` of table `ORDERS` and column `Num` of table `CUSTOMER`. Indeed, each row insertion on table `ORDERS` is preceded by the execution of a validation query that (1) counts the number of rows of table `CUSTOMER` having c as value of column `Num` – where c corresponds to the value of column `Sender` of the inserted row of `ORDERS` – and (2) returns 1 as a result. In other words, it seems that the program checks that the provided value of column `Sender` does correspond to the identifier (`Num`) of an existing customer.

6.9 Initial experiment

An initial experiment was conducted based on an e-learning application, called *WebCampus*[3], that is used at the University of Namur.

6.9.1 The application

WebCampus is an instantiation of *Claroline*[4], an open-source Learning Management System (LMS) allowing teachers to offer online courses and to manage learning and collaborative activities on the web. The platform is written in PHP and manipulates a MySQL database. WebCampus consists of more than a thousand source code files, amounting to 460 thousands lines of code.

The database manipulated by WebCampus consists of two distinct database schemas:

- the main database schema, made up of 33 tables and 198 columns, represents the data on available online courses, course users, university departments, etc. ;
- the course database schema, made up of around 50 tables per course.

The MySQL DDL code of the database does not explicitly declare any foreign key. Indeed, the database makes use of the MyISAM storage engine, which does not support foreign key management. However, the Claroline developers community is aware of all the implicit referential constraints.

[3] see `http://webcampus.fundp.ac.be`

[4] see `http://www.claroline.net`

6.9.2 Goal of the experiment

Our experiment particularly focused on the main database schema of WebCampus, which hides 35 undeclared foreign key constraints. The general goal of the experiment was to evaluate the usefullness of SQL execution traces as a basis for the detection of implicit foreign keys. More precisely, it aimed to check whether it was possible to discover indications of the 35 foreign keys using dynamic analysis of SQL queries.

6.9.3 Methodology

The experiment involved the two following steps:

Step 1. Collecting SQL execution traces corresponding to typical interaction scenarios within WebCampus;

Step 2. Analyzing those traces in relation to the list of implicit foreign keys.

6.9.3.1 Trace collection

The SQL traces collected correspond to the following 14 execution scenarios, which translate typical actions that can be performed on top of the WebCampus platform:

- an administrator creates a course (`create_course`)
- an administrator deletes a course (`delete_course`)
- an administrator tries to delete a referenced department (`delete_dpt_attempt`)
- a course manager adds another manager to a course (`add_course_manager`)
- a course manager adds a user to an course (`add_course_user`)
- a course manager unregisters a user from a course (`delete_course_user`)
- an administrator creates a user account (`register_user`)
- a user registers to WebCampus (`user_register_to_webcampus`)
- a user registers to a course (`user_register_to_course`)
- a user unregisters from a course (`user_unregister_from_course`)
- an administrator installs a tool to the platform (`install_tool`)
- an administrator uninstalls a tool from the platform (`uninstall_tool`)
- an administrator installs an applet (`install_applet`)
- an administrator uninstalls an applet (`uninstall_applet`)

```
CREATE TABLE sql_trace(
  id int(10) unsigned NOT NULL,
  statement text NOT NULL,
  script varchar(128) default NULL,
  scenario varchar(45) NOT NULL,
  PRIMARY KEY  USING BTREE (id, scenario));

CREATE TABLE sql_trace_results(
  id int(10) unsigned NOT NULL,
  id_statement int(10) unsigned NOT NULL,
  row int(10) unsigned NOT NULL,
  column varchar(64) NOT NULL,
  value text NOT NULL,
  call_type varchar(45) NOT NULL,
  scenario varchar(45) NOT NULL,
  PRIMARY KEY USING BTREE (id_statement, scenario, row, column));
```

Figure 6.14: Definition of two tracing tables used to store executed SQL queries and their results.

Trace collection was realized through source code instrumentation. The output of the tracing process was stored in a MySQL database composed of two tables:

- table `sql_trace`, each row of which corresponds to an executed SQL query;
- table `sql_trace_results`, that contains information on the results of those queries.

Figure 6.14 gives the declaration of those two tables in MySQL.

Table 6.2 provides size metrics about the trace obtained by indicating, for each execution scenario, the number and the nature of the corresponding queries and query results.

6.9.3.2 Trace analysis

The trace analysis process started from the schema of the main WebCampus database, augmented with the 35 implicit foreign keys. This schema served as a basis for automatically deriving SQL queries that analyze the contents of the tracing tables. The goal of this analysis is to find indications of the undeclared foreign keys in the execution traces.

Trace reduction and pre-processing The full SQL trace obtained obviously contained queries that are useless for our experiment. This holds for (1) the queries accessing the course database and (2) the queries that do not access the tables involved in an undeclared foreign key. Therefore, the first step of the analysis consisted in subdividing the trace in smaller traces, based on the tables accessed by the queries. For each implicit foreign key from a table t_1 to a table t_2 , two intermediate SQL views were defined:

- view `sql_trace_`t_1`_`t_2, that contains only the queries accessing t_1 and/or t_2;

Execution scenario	**Total # of queries**	**# of queries on main DB**	**# of select on main DB**	**# of insert on main DB**	**# of delete on main DB**	**# of update on main DB**	**Total # of results**	**# of results from main DB**
register_user	27	27	24	3	0	0	163	163
add_course_manager	364	194	190	4	0	0	2 643	2 391
add_course_user	289	155	151	4	0	0	2 112	1 908
create_course	70	29	20	9	0	0	319	299
delete_course	329	132	123	1	7	0	1 865	1 700
delete_course_user	159	84	83	0	1	0	1 110	996
delete_dpt_attempt	37	37	37	0	0	0	423	419
install_applet	92	88	82	4	0	2	729	721
install_tool	4 894	2 169	2 039	126	4	0	26 002	24 180
uninstall_applet	82	78	68	0	9	1	581	573
uninstall_tool	3 713	1 896	1 888	0	8	0	23 333	22 419
user_register_to_course	64	64	63	1	0	0	721	708
user_register_to_webcampus	35	32	30	2	0	0	188	184
user_unregister_from_course	24	19	17	1	1	0	169	155
Total	**10 179**	**5 004**	**4 815**	**155**	**30**	**3**	**60 358**	**56 816**

Table 6.2: Some metrics about the SQL trace obtained, classified by execution scenario.

```
CREATE VIEW sql_trace_t1_t2 as
 SELECT * FROM sql_trace
 WHERE statement like ’%t1%’ or
       statement like ’%t2%’";

CREATE VIEW sql_trace_results_t1_t2 as
 SELECT * FROM sql_trace_results
 WHERE (id_statement, scenario) in (SELECT id, scenario FROM sql_trace_t1_t2);
```

Figure 6.15: Definition of intermediate views for each implicit foreign key (pseudo-code).

- view sql_trace_results_t_1_t_2, that contains the results of those queries.

Figure 6.15 gives the pseudo-code of the definition of those views, where t1 and t2 are, respectively, the source and target tables of an undeclared foreign key.

From those views, several intermediate tables were produced[5], including:

- table augmented_sql_trace_t_1_t_2, that stores the result of the join between sql_trace_t_1_t_2 and sql_trace_results_t_1_t_2;
- table select_statement_t_1_t_2_t1, that contains the select queries on t_1 having an equality condition on the implicit foreign key column fk referencing t_2;
- insert_statement_t_1_t_2_t1, that contains the insert queries on t_1, assigning a value to the implicit foreign key column fk referencing t_2;
- table delete_statement_t_1_t_2_t1, that contains the delete queries on t_1 having an equality condition on the implicit foreign key column fk referencing t_2.

Figure 6.16 gives the definition of the structure of those intermediate tables, as well as data inserted into them.

Trace querying The trace querying step aimed at extracting implicit foreign key indications from the intermediate views and tables defined so far. This trace analysis process made use of SQL queries automatically generated from the main database schema of WebCampus, augmented with the 35 implicit foreign keys. We mainly considered two kinds of foreign key indications, namely joins and output-input dependencies.

The pseudo-query of Figure 6.17 allows to count the number of SQL joins between tables t_1 and t_2 occuring in the SQL trace where the join condition is based on the equality between the implicit foreign key column $t_1.fk$ and the target column $t_2.id$. Figure 6.18 gives a pseudo-query allowing to count the number of output-input dependencies between a query q_1 on table t_2 and a subsequent select

[5] mainly for performance reasons

```
CREATE TABLE augmented_sql_trace_t1_t2(
    id int(10) unsigned NOT NULL,
    statement text NOT NULL,
    script varchar(128),
    scenario varchar(45) NOT NULL,
    row int(10) unsigned NOT NULL,
    column varchar(64) NOT NULL,
    value text NOT NULL,
    call_type varchar(45) NOT NULL,
    PRIMARY KEY USING BTREE (id, scenario, row, column));

INSERT INTO augmented_sql_trace_t1_t2(
   SELECT t.id, t.statement, t.script, t.scenario,
          r.row, r.column, r.value, r.call_type
   FROM  sql_trace_t1_t2 t, sql_trace_results_t1_t2 r
   WHERE t.id = r.id_statement and
         t.scenario = r.scenario);

CREATE TABLE select_statement_t1_t2_t1(
    id int(10) unsigned NOT NULL,
    statement text NOT NULL,
    script varchar(128) default NULL,
    scenario varchar(45) NOT NULL,
    PRIMARY KEY  USING BTREE (id, scenario));

INSERT INTO select_statement_t1_t2_t1(
    SELECT id, statement, script, scenario
    FROM  sql_trace_t1_t2
    WHERE statement like ’SELECT%FROM%t1%’ and
          statement regexp ’(fk)([ ]*)(=)’);

CREATE TABLE insert_statement_t1_t2_t1(
    id int(10) unsigned NOT NULL,
    statement text NOT NULL,
    script varchar(128) default NULL,
    scenario varchar(45) NOT NULL,
    PRIMARY KEY  USING BTREE (id, scenario));

INSERT INTO select_statement_t1_t2_t1(
    SELECT id, statement, script, scenario
    FROM  sql_trace_t1_t2
    WHERE statement like ’INSERT%INTO%t1%’ and
          statement like ’%fk%’);

CREATE TABLE delete_statement_t1_t2_t1(
    id int(10) unsigned NOT NULL,
    statement text NOT NULL,
    script varchar(128) default NULL,
    scenario varchar(45) NOT NULL,
    PRIMARY KEY  USING BTREE (id, scenario));

INSERT INTO delete_statement_t1_t2_t1(
    SELECT id, statement, script, scenario
    FROM  sql_trace_t1_t2
    WHERE statement like ’DELETE%FROM%t1%’ and
          statement regexp ’(fk)([ ]*)(=)’);
```

Figure 6.16: Definition of intermediate tables for each implicit foreign key $t_1.fk \rightarrow t_2.id$ (pseudo-code).

```
SELECT count(statement)
FROM sql_trace_t1_t2
WHERE (statement like '%t1%t2%' or
       statement like '%t2%t1%') and
      (statement regexp '([^ ]*)(\.)(id)([ ]*)([=])([ ]*)([^ ]*)(\.)(fk)' or
       statement regexp '([^ ]*)(\.)(fk)([ ]*)([=])([ ]*)([^ ]*)(\.)(id)')
```

Figure 6.17: Counting foreign-key based joins between tables t_1 and t_2, that occur in the SQL trace (pseudo-code).

```
SELECT count(distinct(concat(concat(q1.id, '-'), q2.id)))
FROM augmented_sql_trace_t1_t2 q1, select_statement_t1_t2_t1 q2
WHERE q2.id > q1.id and
      q2.scenario = q1.scenario and
      q1.column = 'id' and
      q1.statement like '%t2%' and
      q2.statement regexp concat('(fk)([ ]*)(=)([ ]*)([\']*)',q1.value)
```

Figure 6.18: Counting foreign-key based output-input dependencies between a query q_1 on t_2 and a subsequent select query q_2 on t_1 (pseudo-code).

query q_2 on table t_1, where an output value of q_1 corresponding to $t2.id$ is used as input value of q_2 for $t1.fk$.

6.9.4 Results

Table 6.3 indicates, for each implicit foreign key fk from table t_1 to table t_2:

(1) the number of queries referencing t_1;

(2) the number of queries referencing t_2;

(3) the number of distinct scenarios where both t_1 and t_2 are accessed.

From (3), we can derive that only 27 implicit foreign keys (those in light grey) are *potentially detectable* in the SQL trace we obtained. Indeed, the minimal requirement for detecting an undeclared foreign key $t_1 \rightarrow t_2$ in the SQL trace is that both t_1 and t_2 must be involved in at least one execution scenario considered. If this is the case, then the SQL trace obtained *could* contain indications of the foreign key.

Table 6.4 summarizes the indications of implicit foreign key that have been found in the SQL trace by our analyzer. For each undeclared foreign key ($t_1 \rightarrow t_2$), we provide:

(1) the number of SQL joins between t_1 and t_2;

(2) the number of output-input dependencies between a query q_1 accessing t_2 and a subsequent query q_2 accessing t_1, further subdivided according to the nature of q_2.

Implicit foreign key ($t_1 \rightarrow t_2$)	**# q(t_1)**	**# q(t_2)**	**# scenarios(t_1 & t_2)**
`class` → `class`	0	0	0
`cours` → `right_profile`	1927	51	12
`cours_user` → `user`	55	41	9
`cours_user` → `right_profile`	55	51	9
`desktop_portlet_data` → `user`	0	41	0
`FUNDP_user_ADDONS` → `user`	11	41	3
`FUNDP_user_ADDONS` → `FUNDP_program`	11	5	3
`im_message_status` → `user`	4	41	3
`im_message_status` → `im_message`	4	3	3
`im_recipient` → `user`	4	41	3
`im_recipient` → `im_message`	4	3	3
`log` → `user`	3	41	3
`module_contexts` → `module`	38	496	14
`notify` → `user`	9	41	6
`notify` → `course_tool`	9	452	2
`rel_class_user` → `user`	0	41	0
`rel_class_user` → `class`	0	0	0
`rel_course_class` → `class`	1	0	0
`right_rel_profile_action` → `right_profile`	161	51	7
`right_rel_profile_action` → `right_action`	161	178	7
`sso` → `user`	0	41	0
`tracking_event` → `user`	3	41	2
`user_property` → `user`	1	41	1
`cours` → `faculte`	1927	1840	12
`cours_user` → `cours`	55	1927	9
`dock` → `module`	244	496	14
`faculte` → `faculte`	1840	1840	12
`FUNDP_course_program` → `cours`	0	1927	0
`FUNDP_cours_ADDONS` → `cours`	1838	1927	9
`im_message` → `cours`	3	1927	3
`module_info` → `module`	17	496	4
`notify` → `cours`	9	1927	6
`property_definition` → `user_property`	0	1	0
`rel_course_class` → `cours`	1	1927	1
`right_rel_profile_action` → `cours`	161	1927	6

Conventions : q(t) = queries accessing table t;
scenarios(t_1 & t_2) = execution scenarios where both t_1 and t_2 are accessed.

Table 6.3: Number of queries and number of scenarios accessing the tables involved in undeclared foreign keys.

From (1) and (2), we notice that 23 implicit foreign keys (those in light grey) proved to be *detectable* in the trace we obtained, which represents:

- about 65% of the total number of implicit foreign keys in the main database;
- about 85% of the foreign keys identified as *potentially detectable* in the trace we obtained.

6.9.5 Discussion

The experiment presented above clearly confirms that SQL execution traces may contain usefull information about implicit schema constraints, in particular about undeclared foreign keys. In our results, we simply observe that *if there is an implicit foreign key, then the trace probably contains indications of this foreign key*

Implicit foreign key ($t_1 \to t_2$)	**total # joins between t_1 and t_2**	**# output-input dep. q(t_2) → q(t_1)**	**# output-input dep. `select`(t_2) → `select`(t_1)**	**# output-input dep. `select`(t_2) → `insert`(t_1)**	**# output-input dep. `select`(t_2) → `delete`(t_1)**
`class` → `class`	0	0	0	0	0
`cours` → `right_profile`	0	1	0	1	0
`cours_user` → `user`	7	50	44	4	2
`cours_user` → `right_profile`	0	9	0	9	0
`desktop_portlet_data` → `user`	0	0	0	0	0
`FUNDP_user_ADDONS` → `user`	0	15	15	0	0
`FUNDP_user_ADDONS` → `FUNDP_program`	0	0	0	0	0
`im_message_status` → `user`	0	2	0	2	0
`im_message_status` → `im_message`	0	0	0	0	0
`im_recipient` → `user`	0	2	0	2	0
`im_recipient` → `im_message`	0	0	0	0	0
`log` → `user`	0	1	0	1	0
`module_contexts` → `module`	34	11	0	6	5
`notify` → `user`	0	11	11	0	0
`notify` → `course_tool`	0	0	0	0	0
`rel_class_user` → `user`	0	0	0	0	0
`rel_class_user` → `class`	0	0	0	0	0
`rel_course_class` → `class`	0	0	0	0	0
`right_rel_profile_action` → `right_profile`	0	711	91	600	20
`right_rel_profile_action` → `right_action`	32	810	0	810	0
`sso` → `user`	0	0	0	0	0
`tracking_event` → `user`	0	2	0	2	0
`user_property` → `user`	1	0	0	0	0
`cours` → `faculte`	1 832	87	86	1	0
`cours_user` → `cours`	9	47	37	7	3
`dock` → `module`	58	297	291	3	3
`faculte` → `faculte`	3	0	0	0	0
`FUNDP_course_program` → `cours`	0	0	0	0	0
`FUNDP_cours_ADDONS` → `cours`	0	3 842	3 839	0	3
`im_message` → `cours`	0	5	0	5	0
`module_info` → `module`	13	11	0	6	5
`notify` → `cours`	0	3	0	0	3
`property_definition` → `user_property`	0	0	0	0	0
`rel_course_class` → `cours`	0	3	0	0	3
`right_rel_profile_action` → `cours`	0	22	19	0	3

Conventions : q(t) = queries accessing table t;
`select`(t) = `select` queries on table t;
`insert`(t) = `insert` queries on table t;
`delete`(t) = `delete` queries on table t.

Table 6.4: Indications of the implicit foreign keys found in the SQL execution trace.

if both involved tables are accessed. However, the experiment does not allow to claim that *if the trace contains indications of an implicit foreign key constraint, then this constraint probably holds.* In other words, we are not able (yet) to draw conclusions regarding the suitability of the different heuristics (joins, output-input dependencies, input-input dependencies) as a means to discover implicit foreign keys in SQL execution traces. Other experiments are still needed in order to evaluate the amount of *false-negatives* and *false-positives* induced by the use of those heuristics for implicit foreign key detection.

Our SQL trace analysis tool aims to check whether the SQL trace contains indications of *existing* implicit foreign keys. As an interesting side effet, the same tool may also be used for *confirming candidate* implicit foreign keys. For instance, we could discover such candidate foreign keys through an initial schema analysis process, based on the comparison of column names and types. However, our trace analysis tool does not allow (yet) to *detect* implicit foreign keys *from scratch.* To this aim, the tool should consider, for instance, *all* the SQL joins rather than only *some of them.*

Another interesting application of SQL trace analysis concerns the identification of *unsafe data access paths* (Cleve et al., 2008b), i.e., program fragments where an implicit constraint is not correcly managed. For instance, based on the list of implicit foreign keys, one could detect in the SQL trace that an `insert` or an `update` statement is performed without prior verification of the referential constraints. In this case, the analysis would be based on the *absence* of output-input or input-input dependency.

6.10 Conclusions and perspectives

SQL statements appear to be a particularly rich source of information in program and data structure understanding and therefore their analysis must improve such essential processes as program and database maintenance, evolution and reengineering. Though some encouraging results have been obtained in the 90's, particularly to support database reverse engineering, systematically exploring and mastering this information source still is a largely unexplored research and technical domain. The goal of this chapter is, quite modestly, to mark this engineering domain out by identifying and discussing its basic concepts, its specific techniques and some of its representative applications. It particularly provided an in-depth exploration of the use of dynamic program analysis techniques for reverse engineering relational databases. Those techniques particularly target the analysis of data-intensive systems in the presence of *automatically generated* SQL queries.

We first identified, illustrated and compared a set of techniques for *capturing* the SQL queries executed at runtime. Then, we elaborated on the analysis of SQL traces in the context of database reverse engineering. We identified possible heuristics for the detection of implicit foreign key from SQL execution traces. Those heuristics combine both intra-query dependencies (SQL joins) and inter-query dependencies (input-input and output-input dependencies). An initial experiment, based on a real-life application, allowed us to establish the analysis of SQL execution traces as a very promising technique for relational database reverse engineering.

While this chapter basically is exploratory, we have the feeling that the framework we have designed describes adequately this software analysis domain. However, much remains to be done to validate, evaluate and instrument the presented techniques. Three main paths of further research can be identified and will be explored in the future: (1) applying and evaluating our dynamic analysis techniques to other real-world, data-intensive applications; (2) exploring the combination of schema analysis, static analysis and dynamic analysis techniques and (3) developing supporting tools and integrating them in CASE environments.

Roadmap

This chapter concludes the thesis part dedicated to the use of program analysis techniques in support to the initial step of database evolution, namely database reverse engineering. In Part IV, we assume that a legacy database has been migrated towards a modern platform, and we show how to automatically adapt the legacy programs accordingly. Part V is dedicated to another particular database evolution scenario, namely database schema change.

Part IV

Adapting Programs to Database Platform Migration

Chapter 7

Migrating Standard Files to a Relational Database

The use of COBOL cripples the mind; its teaching should, therefore, be reguarded as a criminal offense[1].
– E.W. Dijkstra

In this chapter, we focus on a particular case of database evolution scenario, namely the migration of COBOL applications manipulating standard files towards a relational database platform. We assume that (some of) the files have been migrated to a relational database and we describe how the COBOL programs can be transformed accordingly, so that they access the new relational database instead of the migrated files.

The chapter is organized as follows. Section 7.1 first provides a brief introduction to COBOL file management. Section 7.2 discusses the application of a *Wrapper* program conversion strategy (P1) in the context of the migration of COBOL files towards a relational platform. Section 7.3 briefly elaborates on the *Statement Rewriting* program conversion strategy (P2) in the same context. Systematic translation rules allowing to simulate COBOL file handling primitives on top of a relational database are specified and illustrated in Section 7.4. Those rules are suitable whatever the chosen program conversion strategy. Section 7.6 presents a rapid overview of the tools we have developpped in support to both program conversion strategies. Concluding remarks are given in Section 7.8.

7.1 COBOL file management

7.1.1 File

A COBOL file is an organized collection of related data (Johnson, 1986). A COBOL program can read and write files. Each file is defined in two distinct parts of the

[1] The author obviously disagrees...

```
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 7.1: 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 7.2: Example of `FD` paragraph

program source code (Henrard, 2003):

- 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 7.1 shows an example of such a paragraph, also called ”`SELECT` clause”.

- The `FD` paragraphs of the `FILE` section of the `DATA` division declare the record types, their field decomposition and the type and the length of the fields. Figure 7.2 shows an example of `FD` paragraph.

7.1.2 Record type

A COBOL file is made up of records. A record is a collection of related data items treated as a single unit. The record type associated with a file consists of the set of indivisible data, read or written during the access to that file. The data items that make up a record type are called fields. In the case of file `ORDERS` in Figure 7.2, the record type is `ORD`. Actually, the record type of a COBOL file f corresponds to the data item declared at level 01 in the `FD` paragraph of f.

7.1.3 File organization

A COBOL file can be organized according to three different ways (Brown, 1998):

- `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 based on a given key value.

7.1.4 File access mode

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

- SEQUENTIAL (default), according to which the records are read or written sequentially;
- RANDOM, that requires the programmer to supply a key to read or write a record;
- DYNAMIC, which allows the programmer to read the file with both SEQUENTIAL and RANDOM accesses, while record writing is performed on a RANDOM basis.

7.1.5 Access keys

For each SEQUENTIAL or INDEXED file, the programmer declares a RECORD KEY, that is one of the fields allowing to uniquely identify each record. In our example of Figures 7.1 and 7.2, field ORD-CODE is declared as the RECORD KEY of file ORDERS.

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 (Clarinval, 1981).

The *primary key* of a file is the data item used to identify each record in the file. This identifier corresponds to:

- The RECORD KEY, for a sequential or an indexed file.
- The RELATIVE KEY, for a relative file.

Other data items used as keys are called ALTERNATE RECORD KEYs. These keys 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 7.1, for record key ORD-CUSTOMER and ORD-DATE.

7.1.6 DML statements

The main COBOL file handling primitives are the following:

- The OPEN statement, that allows to open a file in reading and/or writing mode;
- The START statement, that positions the reading sequence on a given record;
- The READ statement, that allows to access the records sequentially or randomly;
- The WRITE statement, that inserts new records in a file;
- The REWRITE statement, used to update records of a file;

- The DELETE statement, allowing to discard records from a file;
- The CLOSE statement, that closes the files and makes them available for processing by another application.

A precise definition of the syntax and effect of each file handling primitive is provided in Appendix A.

7.2 Wrapper-based program conversion

As seen in Chapter 4, the *wrapper-based* program conversion strategy (P1) assumes that a wrapper encapsulates the new database obtained through the database conversion step. The P1 conversion strategy applied to COBOL programs mainly consists in replacing the COBOL DML statements (accessing the files that have been migrated) with a corresponding wrapper invocation. In addition, other parts of the programs must be reorganized like, for instance, the declaration of the migrated files and record types.

A COBOL program is composed of four main *divisions*:

- The IDENTIFICATION division, containing comments identifying the program, its author, and the date it was written.
- The ENVIRONMENT division, that gives a name to the source and object computer and describes each file used by the program.
- The DATA division, that declares all the data items manipulated by the programs (record types or variables).
- The PROCEDURE division, which regroups the executable program statements.

The wrapper-based program conversion strategy affects the last three divisions. Below, we describe more precisely what these divisions contain, and how they can be modified according to wrapper-based strategy.

7.2.1 Environment division

The ENVIRONMENT division, an example of which is given in Figure 7.3, consists of two optional sections: the CONFIGURATION section and the INPUT-OUTPUT section. The latter regroups the file definitions. Each file corresponds to a SELECT clause, that associates the external name of the file with the name used to reference it in the program. This association is made through the ASSIGN clause. The SELECT clause may also provide information on the file organization, the file access mode, the corresponding record key and alternate record keys.

Once the programs have been transformed, they do not access the migrated files anymore. Thus, the SELECT clauses declaring those files 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 7.3: Sample ENVIRONMENT division.

7.2.2 Data division

The DATA division consists of the main following sections:

- The FILE section: where the record structure of files is defined;
- The WORKING-STORAGE section: that describes all data items belonging to the program;
- The LINKAGE section: which describes the invocation parameters of the program.

Two modifications of the DATA division are required. First, the record type definitions of the migrated files have to be moved from the FILE section to the WORKING-STORAGE section. Second, several new variables have to be declared in the WORKING-STORAGE section. Figure 7.4 gives an example of a DATA division adaptation. In this example, three files CUSTOMER, ORDERS and STOCK are considered for migration. The three corresponding record type definitions (CUS, ORD and STK), initially located in the FILE section (lines [4-7], [10-14] and [17-20] of Figure 7.4a), are moved to the WORKING-STORAGE section (lines [3-6], [8-12] and [14-17] of Figure 7.4b). 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 useless.

7.2.3 Procedure division

The wrapper-based program conversion strategy consists in replacing each legacy DML statement with a corresponding wrapper invocation. In COBOL, calling an external program is performed through the execution of a CALL statement. Since

```
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 7.4: Sample DATA division transformation.

```
01  WR-ACTION  PIC 99.
   88  WR-ACTION-OPEN VALUE 0.
   88  WR-ACTION-WRITE VALUE 1.
   88  WR-ACTION-READ VALUE 2.
   88  WR-ACTION-START VALUE 3.
   88  WR-ACTION-REWRITE VALUE 4.
   88  WR-ACTION-DELETE VALUE 5.
   88  WR-ACTION-CLOSE VALUE 6.

01 WR-OPTION PIC X(100).

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 2.
   88 WR-STATUS-SQLCODE VALUE 3.
```

Figure 7.5: COBOL definition of the wrapper invocation arguments.

we consider one wrapper per migrated record type, the wrapper to be invoked depends on the file accessed by the legacy DML statement. A wrapper invocation in COBOL has the following general form:

`CALL` *wrapper-name* `USING` *wr-action*, *record-name*, *wr-option*, *wr-status*

where:

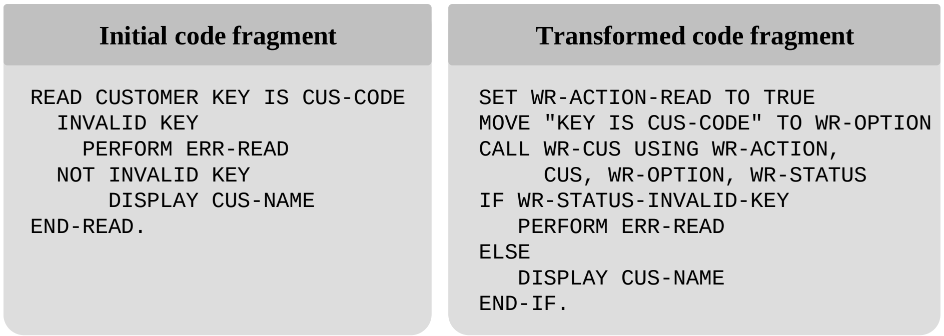
- *wr-action* specifies the type of access to perform;
- *record-name* is the COBOL record type involved in the operation;
- *wr-option* specifies an option associated with the operation;
- *wr-status* indicates whether the operation has been successfully performed.

Such an invocation causes a given wrapper to simulate a particular file handling primitive (*wr-action*) related to a given record type (*record-name*) on top of the relational database, and according to a given option (*wr-option*). The first, third and fourth arguments of the wrapper invocation must be declared as new variables in the `WORKING-STORAGE` section. Figure 7.5 shows the COBOL definitions of these new variables.

The replacement of COBOL file handling primitives with wrapper invocations is illustrated in Figure 7.6. A random read statement is rewritten as a code fragment that invokes the corresponding wrapper and simulates the initial `INVALID KEY` and `NOT INVALID KEY` phrases based on the resulting value of `WR-STATUS`.

7.3 Statement rewriting program conversion

The *Statement rewriting* program conversion strategy (P2) aims at replacing each COBOL file handling primitive with a code fragment translating this primitive on top of SQL.



Initial code fragment

```
READ CUSTOMER KEY IS CUS-CODE
  INVALID KEY
    PERFORM ERR-READ
  NOT INVALID KEY
      DISPLAY CUS-NAME
END-READ.
```

Transformed code fragment

```
SET WR-ACTION-READ TO TRUE
MOVE "KEY IS CUS-CODE" TO WR-OPTION
CALL WR-CUS USING WR-ACTION,
     CUS, WR-OPTION, WR-STATUS
IF WR-STATUS-INVALID-KEY
   PERFORM ERR-READ
ELSE
   DISPLAY CUS-NAME
END-IF.
```

Figure 7.6: Replacement of a random READ statement with a wrapper invocation.

Similarly to the P1 strategy, the P2 strategy also necessitates the adaptation of the ENVIRONMENT and DATA divisions. This adaptation consists in reorganizing the file and data declarations in these divisions.

According to our approach, the conversion of the PROCEDURE division is performed in two steps:

1. The SQL translation of each possible COBOL DML statement is specified as an additional procedure of the legacy program.

2. Each occurence of a COBOL DML statement is replaced with the invocation of the corresponding procedure, through a PERFORM statement.

For instance, DELETE ORDERS is replaced with PERFORM DELETE-ORDERS", where DELETE-ORDERS is a generated procedure that executes the corresponding SQL delete statement. This approach allows to preserve the readability of the converted code, since the SQL-based code fragments are not duplicated multiple times in the target program.

7.4 COBOL-to-SQL translation

We will now further define how the COBOL file handling primitives can be translated into SQL statements. The provided translation rules are valid whatever the program conversion strategy chosen (*Wrapper* (P1) and *Statement rewriting* (P2)). The main difference between both strategies is the target location of the generated SQL-based code (external program for P1, internal procedure for P2).

7.4.1 Overview

Figure 7.7 briefly summarizes the SQL statements involved in the translation of COBOL DML statements.

COBOL statements	SQL statements involved
OPEN	OPEN cursor
OPEN (output)	DELETE, OPEN cursor
START	SELECT, OPEN cursor
READ (sequential)	FETCH cursor
READ (random)	SELECT, OPEN, FETCH cursor
WRITE	INSERT
REWRITE	UPDATE
DELETE	DELETE
CLOSE	–

Figure 7.7: SQL statements involved in the translation of COBOL file handling primitives.

- Translating an OPEN statement consists in opening the cursor that selects all the rows of the table in the ascending order based on the primary key. If used with the OUTPUT mode, the corresponding SQL table must be emptied beforehand.
- A START KEY IS statement is simulated by opening the cursor corresponding to the specified record selection criteria.
- Simulating the READ NEXT statement involves a FETCH on the last accessed cursor.
- A random READ KEY IS can be seen as a READ NEXT that immediately follows a START KEY IS EQUAL. So a random READ can be simulated by (1) opening the corresponding cursor and (2) executing a FETCH on this cursor.
- A WRITE statement is replaced with a SQL INSERT statement.
- A REWRITE statement is simulated using a SQL UPDATE statement.
- A COBOL DELETE statement becomes a SQL DELETE statement.
- There is no SQL translation for the COBOL CLOSE statement.

7.4.2 Preliminaries

One-to-one mapping

For the sake of clarity, we will consider a one-to-one mapping between the source COBOL schema and the target relational schema (i.e., a D1 schema conversion strategy). In principle, the D1 conversion strategy translates each COBOL record type into a relational table and each top-level field into one column of this table, thereby ignoring its possible decomposition. Figure 7.8 shows an example of such a one-to-one translation. COBOL file ORDERS, declared in Figure 7.3, is converted

```
 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.
...
 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 7.8: One-to-one translation (D1) of a COBOL file into a relational table

```
SELECT STUDENT
  ASSIGN TO "c:\student"
  ORGANIZATION IS INDEXED
  ACCESS MODE IS DYNAMIC
  RECORD KEY IS NAME
  ALTERNATE RECORD KEY IS LAST-NAME
    WITH DUPLICATES
```

```
FD STUDENT.
01 STUD.
  02 NAME.
    03 FIRST-NAME PIC X(12).
    03 LAST-NAME PIC X(16).
  02 ADDRESS.
    03 STREET PIC X(20).
    03 NUMBER PIC X(5).
    03 CITY PIC X(10).
  02 SCHOOL PIC X(20).
```

Figure 7.9: Definition of file STUDENT.

into table ORD. We can observe that this conversion does not affect the data structure, and preserves the access keys declaration. The primary key of file ORDERS (ORD-CODE) is translated into a column declared as primary key of table ORD. The alternate keys ORD-CUSTOMER and ORD-DATE are translated into two indexes defined on table ORD. The columns are required to be not null since the COBOL fields they translate must always have a value in each record (especially for the access keys).

However, the above example corresponds to the *default* D1 conversion scheme. Let us now consider file STUDENT declared in Figure 7.9. In this particular case, the D1 *default* rule should not be applied. Translating the three top-level fields into three relational columns would prevent to declare sub-level field LAST-NAME as an access key. Therefore, when a top-level field contains a sub-level field defined as an access key, it is *disaggregated* before being translated into columns. The requirement is that each COBOL access key must correspond to one or several relational columns, in order to permit its declaration as an access key. In our example of Figure 7.9, record description STUD would be translated into a table made of four columns, as shown in Figure 7.10. The top-level field NAME has been disaggregated. Its two sub-level fields have been translated into two corresponding

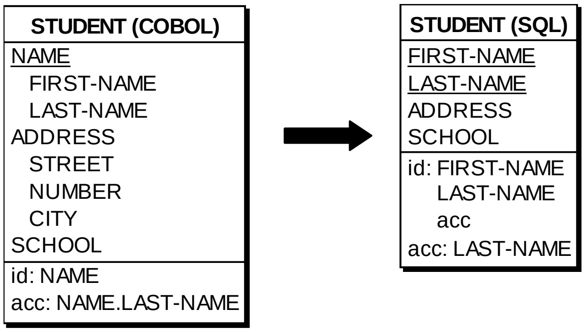


Figure 7.10: Example of a one-to-one schema conversion (D1).

columns. This allows to declare both the record key and the alternate key as indexes in the relational DDL code.

More formally, the one-to-one mapping holding between the legacy COBOL record types and the target SQL tables can be defined as follows. Let $\mathcal{S}_{cob}$ be the source COBOL schema. Let $\mathcal{S}_{rel}$ be the target relational schema. Let $file$ be a COBOL file of $\mathcal{S}_{cob}$. Let *record* be the record type associated with $file$. We then assume that:

- *table* is the relational table of $\mathcal{S}_{rel}$ translating record type *record*.
- *table* contains n columns, called $c_1, c_2, ..., c_n$, that translate fields $f_1, f_2, ..., f_n$ of *record*. When $file$ is a relative file, one column c_i translates the `RELATIVE KEY`.
- $file$ has one primary key, denoted by *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 several columns of *table*.

Help variables

Our COBOL-to-SQL translation rules will make use of several variables, that we briefly describe below.

- `SQLCODE`: is one of the fields of the `SQLCA` structure. It is used as a status returning value for each executed SQL operation. For instance, when its value is equal to zero, it means that the operation was performed successfully.
- `STATUS`: is used to simulate the exceptions associated the COBOL file handling primitives. The `INVALID KEY` and `AT END` exceptions are associated

to a particular value of variable `STATUS`, as shown in Figure 7.5. In such a way, the `INVALID KEY` and `AT END` phrases can be replaced with a conditional statement based on the value of `STATUS`.

- ”`LAST-CURSOR-`t : allows to keep track of the last SQL cursor open on a particular table t. This runtime information is needed in order to correctly simulate a `START/READ NEXT` sequence through a cursor-based loop.

SQL cursors

Below, we specify the SQL cursors that are used for simulating COBOL file handling primitives in SQL. For each access key (*prim-key* or ak_i), three SQL cursors are declared: *order by*, *greater than* and *not less*.

***order by* cursor** The *order by* cursor has the following general form:

```
EXEC SQL
  DECLARE $cursor\text{-}name$ 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].

Note that the ck_1,ck_2,...,ck_p appearing in the `order by` clause should be correcly ordered with respect to the COBOL access key they translate As an example, here below is the ”*order by*” cursor declaration for the access key `NAME` described in Figure 7.9. In this case, the access key has been split into two columns `FIRST_NAME` and `LAST_NAME` ($p = 2$):

```
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 $cursor\text{-}name$ 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
                  ...
```

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

```
            (($ck_1$ = $fk_1$) ... AND ($ck_{p-1}$ = $fk_{p-1}$) AND ($ck_p$ > $fk_p$))
    ORDER BY $ck_1$, $ck_2$,..., $ck_p$
END-EXEC
```

As an example, here below is the *greater than* cursor corresponding to Figure 7.9.

```
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
```

Note that the same COBOL field names can be used in several record type descriptions. For instance, a another field `FIRST-NAME` could be defined 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. For instance, in the case of the example above, we should replace `":FIRST-NAME"` with `":STUD.NAME.FIRST-NAME"`. However, we will not use the complete names in our examples, in order to make them more readable.

***not less* cursor** The *not less* cursor is similar to the *greater than* cursor, but the ”`>`” of the last alternative is replaced with ”`>=`”:

```
EXEC SQL
  DECLARE $cursor\text{-}name$ 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
```

Current cursor closing For each relational table t, a COBOL procedure that closes the current open cursor associated with t (`LAST-CURSOR-`t), if any. Figure 7.11 shows an example of such a paragraph in the case of the relational table `STUD` in Figure 7.10.

7.4.3 `OPEN` statement

We have seen that the result of a COBOL `OPEN` statement depends on the opening mode, which can be `INPUT`, `OUTPUT`, `I-O` or `EXTEND`.

There is an important difference between the `OUTPUT` mode and the other modes. When a COBOL file is opened with the `OUTPUT` mode, it is created (if necessary)

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

Figure 7.11: A COBOL procedure that closes the current cursor associated with table STUD.

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

Figure 7.12: `OPEN` translation for file `STUDENT`.

```
OPEN-OUTPUT-STUDENT.
   PERFORM 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 7.13: `OPEN OUTPUT` translation for file `STUDENT`.

and positioned to its starting point for writing. In other words, if the file already exists, it is overwritten. The other opening modes allow the file to be either read or updated.

Another important aspect to take into account is the file access mode. When the access mode is `SEQUENTIAL`, the records are read in the ascending order of 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 specifies any access key. In summary, in all the cases, the first `"READ` *file*`"` statement executed by the program retrieves the record of the file having the lowest primary key value. Therefore, a COBOL `OPEN` statement can be simulated by opening the *order by* cursor associated with the primary key of the target table. When the open option is `OUTPUT`, the contents of the table are deleted. Figure 7.12 shows the COBOL procedure that simulates the `OPEN` statement for file `STUDENT` discussed above. Figure 7.13 illustrates the SQL translation of the `OPEN OUTPUT` statement on the same example.

7.4.4 `START` statement

As seen above, the `START` statement allows to position an indexed or relative file to a specific record. In SQL, this can also be simulated by opening a cursor 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` statement may also specify an `INVALID KEY` phrase, the translation starts by checking whether there exists at least one row in the table where 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, an `INVALID KEY` exception must be simulated, based on variable `STATUS`. If there

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

Figure 7.14: `START` key usages VS SQL cursors.

```
START-STUDENT-GREATER-NAME.
   PERFORM CLOSE-LAST-CURSOR-STUD.
   EXEC SQL
      SELECT COUNT(*)
      INTO :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 STATUS-SQL-ERROR TO TRUE
   ELSE
      IF (COUNTER = 0) %% invalid key
         SET STATUS-INVALID-KEY TO TRUE
      ELSE
         EXEC SQL
            OPEN GREATER_NAME
         END-EXEC
         MOVE "GREATER-NAME" TO LAST-CURSOR-STUD
         MOVE SQLCODE TO STATUS
      END-IF
   END-IF.
```

Figure 7.15: `START` translation for file `STUDENT`.

is at least one such row, the corresponding cursor can be opened. Figure 7.14 indicates the SQL cursor that is opened for each `START` key usage. Figure 7.15 illustrates the simulation of the `START` *key is greater than* statement in SQL, in the case of file `STUDENT` of Figure 7.9.

7.4.5 `READ NEXT` statement

The sequential `READ` statement can be translated into a SQL `FETCH` statement executed on the current cursor of the corresponding table. Based on the value of the `LAST-CURSOR` variable, the name of the current open cursor can be determined. The `INTO` clause of the `FETCH` statement contains the COBOL record fields that have been translated into SQL columns $(f_1, f_2, ..., f_n)$, in the correct order.

Figure 7.16 shows the procedure translating the `READ NEXT` statement for the file `STUDENT` described above. The last statement of this procedure (`MOVE SQLCODE TO STATUS`) allows to simulate the `AT END` exception. Indeed, if the end of the cursor is reached when executing the fetch statement[3], the resulting value of SQLCODE is

[3] which corresponds to an *end of file* exception

```
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_NAME")
        EXEC SQL
          FETCH GREATER_LAST_NAME
          INTO :FIRST-NAME, :LAST-NAME, :ADDRESS, :SCHOOL
        END-EXEC.
    IF(LAST-CURSOR-STUD = "NOT_LESS_NAME")
        EXEC SQL
          FETCH NOT_LESS_LAST_NAME
          INTO :FIRST-NAME, :LAST-NAME, :ADDRESS, :SCHOOL
        END-EXEC.
    MOVE SQLCODE TO STATUS.
```

Figure 7.16: READ NEXT translation for file STUDENT.

equal to 100. Assigning this value to the STATUS variable causes the STATUS-AT-END boolean flag to be set to TRUE.

7.4.6 READ KEY IS statement

As already mentioned, the ramdom READ statement can be seen as a START statement with an *equal* key usage, combined with a READ NEXT. So, the SQL translation of a READ KEY IS statement consists in opening the *not less* cursor before fetching it. Obviously, one first need to check that an INVALID KEY exception does not hold based on a SELECT COUNT query, as shown in Figure 7.17.

7.4.7 WRITE statement

Simulating the WRITE statement in SQL simply consists in inserting a new row in the table that translates the record type of interest, using an INSERT query. Figure 7.18 gives an example of such a WRITE procedure for file STUDENT described above.

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

Figure 7.17: `READ KEY IS` translation for file `STUDENT`.

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

Figure 7.18: `WRITE` translation for file `STUDENT`.

```
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 STATUS.
```

Figure 7.19: `REWRITE` translation for file `STUDENT`.

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

Figure 7.20: `DELETE` translation for file `STUDENT`.

7.4.8 `REWRITE` statement

The translation of the `REWRITE` statement consists of a SQL update of the record (i.e., the row) with the *current* primary key value. This behaviour is valid for both sequential and random access. In the case of sequential access mode, the record must be read before it can be rewritten. In random access mode, the programmer must assign a value to the primary key field before the `REWRITE` statement can be issued. In Figure 7.19, we illustrate the SQL translation of the `REWRITE` statement in the case of file `STUDENT` described above.

7.4.9 `DELETE` statement

The `DELETE` statement can be easily translated into a SQL delete query. The record (i.e., the row) to be deleted is the one having the same primary key value as the current record. This is true for both sequential and random variants of the `DELETE` statement. In sequential access mode, the record must be read before it can be deleted. In random access mode, the programmer must assign a value to the primary key field before the delete statement can be issued.

7.5 About correctness

The translation rules presented above aim at simulating the different COBOL file handling primitives by means of corresponding SQL queries. Each translation rule actually replaces *one* primitive *p* by a *sequence s* of statements. Thus, in order to

NAME		ADDRESS			SCHOOL
FIRST-NAME	LAST-NAME	STREET	NUMBER	CITY	
Vincent	Englebert	Metacase Boulevard	32	Berck	FUNDP
Jean-Luc	Hainaut	Relational Street	17	Namur	FUNDP
Jean-Marie	Jacquet	Linda Avenue	631	Nivelles	FUNDP
Ralf	Lämmel	Grammar Square	9	Koblenz	GTTSE
Kim	Mens	Intensive Street	74	Brussels	UCL
Wim	Vanhoof	Mercury Avenue	319	Leuven	FUNDP

Figure 7.21: Example contents for file STUDENT.

FIRST_NAME	LAST_NAME	ADDRESS			SCHOOL
Vincent	Englebert	Metacase Boulevard	32	Berck	FUNDP
Jean-Luc	Hainaut	Relational Street	17	Namur	FUNDP
Jean-Marie	Jacquet	Linda Avenue	631	Nivelles	FUNDP
Ralf	Lämmel	Grammar Square	9	Koblenz	GTTSE
Kim	Mens	Intensive Street	74	Brussels	UCL
Wim	Vanhoof	Mercury Avenue	319	Leuven	FUNDP

Figure 7.22: Table STUDENT obtained from the file of Figure 7.21.

show that a translation rule is correct, we have to demonstrate that s, seen as a black box, has the same behaviour as p in all cases.

Formally proving the correctness of the translation rules would necessitate to define both the formal semantics of COBOL and the formal semantics of Embedded SQL. Due to space and time limitations, we will rather try to convince the reader, based on the small running example of Figure 7.10, that our translation rules are behaviour-preserving.

Let us consider the example file STUDENT depicted in Figure 7.21, that has been migrated to a relational table STUDENT given in Figure 7.22.

OPEN statement The rule allowing to translate the OPEN STUDENT statement is quite trivial. It mainly consists in opening the *order_by_name* cursor, which selects all the rows from table STUDENT in ascending order of their FIRST_NAME and LAST_NAME columns. In the case of an OUTPUT file opening mode, the content of the relational table is emptied beforehand. The open cursor is then stored as the *current cursor* for table STUDENT. This means that the very same cursor would be fetched in order to simulate the execution of a subsequent READ STUDENT NEXT statement.

START statement For assessing the correctness of a START statement, we have to show that the target source code (1) correctly simulates potential *invalid key* exceptions; and (2) always positions the reading sequence on the right record, in the absence of invalid key exception. Regarding (1), the way of detecting an invalid key exception is to count the number of rows in the table that respect the selection criteria of the START statement. For instance, in the case of a START STUDENT KEY IS > NAME, we will count the number of rows in table STUDENT, for which the combined value of columns FIRST_NAME and LAST_NAME is greater than the value of field NAME of the record buffer. If this number is equal to zero, it means that there exists no remaining rows in the reading sequence, which causes an invalid key exception to be simulated through variable STATUS. Regarding requirement (2), we can observe that the target code always positions the reading sequence on the first row of the table respecting the selection criteria (if any). This is done by opening a corresponding SQL cursor (see Figure 7.14) in which the rows are ordered according to the value of the provided access key. For instance, if the current value of field NAME is "Jean-Luc Hainaud", the reading sequence would be positionned on the student whose name is "Jean-Luc Hainaut". This corresponds to the desired behaviour. In this example, the open *greater-than-name* cursor is then stored as the *current cursor* for table STUDENT.

READ NEXT statement The translation of a READ NEXT statement is correct only if the target source code (1) correctly simulates potential *at end* exceptions; and (2) always retrieve the next record with respect to the current reading sequence, if such a record exists. This behaviour is correctly implemented by our translation rule, since the concept of *current reading sequence* is represented by the *current cursor* of the table. For instance, the translation of a READ STUDENT NEXT primitive simply consists in fetching the current cursor associated to table STUDENT. If there is no remaining row to be fetched, the execution of the fetch statement returns 100 as value of SQLCODE. In this case, an AT END exception is simulated.

READ KEY IS statement Since the READ KEY IS corresponds to the combination of a START KEY IS = statement followed by a corresponding READ NEXT statement, the way to show that the translation rule is correct is to demonstrate that the target code simulating a READ x KEY IS y statement is equivalent to the code simulating the following fragment:

```
START x KEY IS = y
READ x NEXT
```

The analysis of the READ KEY IS translation rule for our running example (Figure 7.17) reveals that this is the case.

WRITE statement The translation rule associated to the WRITE statement is quite trivial. It simply consists in inserting a new row in the target table, from the current value of the corresponding record buffer. In case the specified value of the record key corresponds to an existing row of the table, the INSERT statement fails and the returned value of SQLCODE is different from zero, based on which an *invalid key* exception can simulated.

REWRITE statement The REWRITE statement modifies at most one row in the target table. This row is the one having the same record key value as the record buffer. If such a record does not exist, the UPDATE statement fails, which in turn results in a SQLCODE value different from zero. In the latter case, an *invalid key* exception is simulated.

DELETE statement The SQL translation of a DELETE statement is similar to the one of the REWRITE statement, except that at most one row is *removed* from the table. The deleted row, if any, is the one having the same record key value as the record buffer. If no such record can be located, an *invalid key* exception is returned.

7.6 Tool support

In this section, we give a brief overview of the tools allowing to support the P1 and P2 program conversion strategies in the context of COBOL-to-relational database migration. The tool architecture (see Figure 7.23) is based on the combination of DB-MAIN (DB-MAIN, 2006) and the ASF+SDF Meta-Environment (van den Brand et al., 2001).

Schema mapping management The propagation of the schema mappings is managed by DB-MAIN during the successive transformations applied to the schema. The schema mapping relationships are expressed by means of schema annotations. A dedicated mapping assistant has been implemented as a Java plugin of DB-MAIN. This assistant allows the user to define and visualize the structural correspondences that exist between two database schemas. In the context of migration the mapping assistant can also be used to validate the (automatically derived) mapping and to adapt it when necessary.

Wrapper and procedure generation The wrapper and procedure generators are DB-MAIN plug-ins developed in Voyager 2[4]. Both code generators take as inputs (1) the legacy COBOL schema, (2) the target relational schema, and (3) the mapping between these two schemas. The wrapper generator produces a set of wrappers, each corresponding to a migrated COBOL record type. The generated wrappers are COBOL programs including embedded SQL commands. The procedure generator produces a COBOL *copybook*[5] that regroups all the procedures allowing to simulate the COBOL file handling primitives on top of the relational database.

[4] Voyager 2 (Englebert, 2002) is a 4th-generation, semi-procedural language which offers predicative access to the repository of DB-MAIN.

[5] a copybook is a kind of include file.

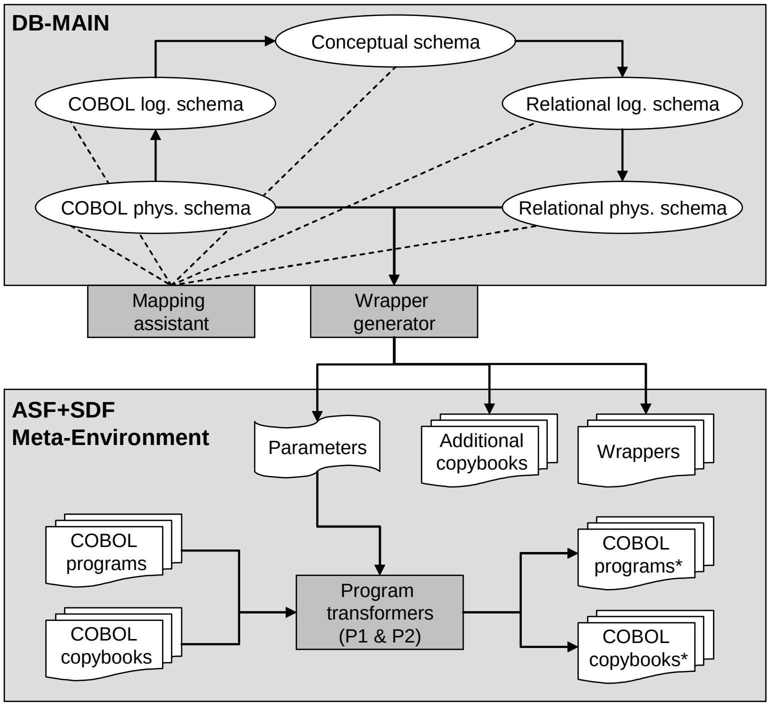


Figure 7.23: Tool architecture for program adaptation in standard files to relational migration.

	Application 1		Application 2	
	Source	Target	Source	Target
# Record types/tables	3	5	8	18
# Fields/columns	10	18	291	276
# Foreign keys	-	5	-	15
# Programs	1	1	15	15
# Wrappers	-	3	-	8
# Wrapper calls	-	26	-	365
Legacy code size (KLOC)	0.4	0.5	7	8.2
Wrapper code size (KLOC)	-	1.6	-	6.1

Figure 7.24: Case studies overview.

Program transformation The program transformation tools are written on top of the ASF+SDF Meta-Environment (van den Brand et al., 2001). We reused and adapted an SDF version of the IBM VS COBOL II grammar obtained by Lämmel and Verhoef (2001). We specified a set of rewrite rules (ASF equations) on top of this grammar in order our main program transformation tool. The rewrite rules make an intensive usage of *traversal functions* van Den Brand et al. (2003). The transformation tool chain also comprises lexical processors (typically Perl scripts) for source-code preprocessing, post-processing and pretty-printing.

7.7 Initial case studies

The approach and tools presented in this chapter have been successfully used to reengineer two small but realistic COBOL applications. Figure 7.24 summarizes the results obtained in those initial case studies, for which we used a wrapper-based program conversion strategy (P1). The first application we migrated corresponds to a variant of the running example used in Chapter 4. In this small example, the application is made up of a single COBOL program manipulating three files, without explicit links between them. The reverse engineering phase allowed us to significantly enrich the legacy schema with (1) finer-grained record type decompositions and (2) implicit foreign keys between the record types. The target relational schema obtained was composed of 5 tables and 5 explicit foreign keys. The file contents were migrated towards the relational database and three corresponding wrappers were automatically generated. The last step consisted in interfacing the small COBOL program with the wrappers, by means of our program transformation tool.

As a second experiment, we followed the same tool-supported approach to convert an academic COBOL application that was in use at the University of Namur. This application, made up of 15 programs and about 7 000 lines of code, managed 8 large files containing data about students, registrations, results of exams, etc. The migration of those files resulted in a target relational database comprising 18 tables. The legacy programs were then interfaced with 8 generated wrappers, the latter totaling more than 6 000 lines of COBOL code.

For both case studies, the program conversion phase (wrapper generation + legacy code transformation) was fully automated. As expected, we observed a small increase of the legacy source code size. Indeed, each wrapper invocation typically necessitates several lines of code and the rewritten statements are preserved as comments in the target programs. Systematic tests confirmed that (1) the wrappers were correcly generated, (2) the programs were correcly transformed and (3) the converted programs had the same behaviour as the original programs.

7.8 Conclusions

In this chapter, we have studied the automated adaptation of legacy programs in the context of the migration of COBOL files towards a relational database. We have described the instantiation of the *wrapper* (P1) and *statement rewriting* (P2) strategies, presented in Chapter 4, in this particular migration scenario. A set of COBOL-to-SQL conversion rules have been proposed and illustrated. A set of tools supporting the program adaptation process have been presented.

The main conclusion we can draw from this chapter is that the combination of transformational and generative techniques provides a sound basis for legacy source code migration. The generated code sections, which are not supposed to be manually maintained, capture the major complexity of the conversion process which originates from the database paradigm change. We also observed that the more significant difference between the P1 and P2 strategies concerns the way of interfacing the legacy programs with the generated code fragments (program code sections or wrappers).

Roadmap

In Chapter 8, we will address another popular, yet much more challenging system migration scenario, that is, the conversion of a CODASYL database towards a relational platform. In particular, we will present a tool-supported, wrapper-based approach for adapting the legacy programs to the target database, i.e., for *simulating* CODASYL data manipulation statements on top of a relational database. Chapter 9 then presents the application of the approach and tools in support to industrial database migration projects.

Chapter 8

Migrating a CODASYL Database to a Relational Database

Success is to be measured not so much by the position that one has reached in life as by the obstacles which he has overcome.
– Booker T. Washington

This chapter[1] describes a generic methodology to migrate all the components of a large software system due to database porting towards a modern platform. This methodology is discussed through a popular specific scenario, that is, the migration of a large COBOL system based on a CODASYL DBMS to a relational database. The chapter shows how a wrapper-based approach can minimize the program adaptation effort. It analyzes the main challenges posed by the simulation of the CODASYL API on top of the relational interface.

The chapter is organized as follows. Section 8.1 briefly presents the basic concepts of CODASYL database management. In Section 8.2 we summarize our global migration methodology. Section 8.3 provides an in-depth analysis of the wrapper-based approach to program conversion. In Section 8.4, we propose systematic translation rules allowing to simulate CODASYL database statements in SQL. Section 8.5, describes a set of tools providing an automated support to most migration processes of our approach. A related work discussion is provided in Section 8.6 and Section 8.7 concludes the chapter.

8.1 CODASYL data management

This section presents an overview of CODASYL data management systems. For further details about the network data model we refer to the overviews by Elmasri and Navathe (1999) and Hainaut (2009), from which the present section is inspired.

[1] An earlier version of this chapter was published in the Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008) in April 2008 (Cleve et al., 2008a).

8.1.1 General DBMS architecture

CODASYL DBMSs typically include the three following components:

- the database control system (DBCS), which executes orders received from the application programs.
- the DDL compiler, which translates the data description code into internal tables that are stored in the database, so that they can be exploited by the DML compiler at program compile time and by the DBCS at program run time.
- the DML compiler, that parses applications programs and replaces DML statements with calls to DBMS procedures. Either this compiler is integrated into the host language compiler (typically COBOL) or it acts as a precompiler.

Each program has its own *user working area* (UWA) in which data to and from the database are stored.

8.1.2 Records, record types and fields

CODASYL databases are made of *records*, each consisting of a group of data values. Records are classified into *record types*. Each record type describes the structure of a group of records storing the same type of information. We give each record type a name, and we also give a name and format for each *field* (a.k.a. *data item*) of the record type.

8.1.3 Areas

An area (a.k.a. *realm*) is a named logical repository of records of one or several types. An area may contain records of more than one record type. Converserly, records of the same record type may be stored in more than one area. Areas provide a way to partition the set of the database records either logically (e.g., according to geographic, organizational or temporal dimensions) or physically (e.g., across disk drives).

8.1.4 Set types

A *set type* describes a one-to-many relationship between two record types: an *owner* record type and a *member* record type. In the database itself, there can be many *set instances* (a.k.a. *set occurences*) corresponding to a given set type. Each set instance is composed of:

- one owner record from the owner record type;
- zero or more member records from the member record type.

As indicated above, a set type represents a one-to-many relationship. Consequently, for a given set type, a record from the member record type cannot belong to more than one set instance.

Note that a set instance in CODASYL differs from the concept of set in mathematics. First, the set instance typically contains two kinds of elements: the owner record and the member records. Second, the member records of a set instance are *ordered* according to a user-specified ordering criteria. Such an order of elements is not considered in a mathematical set.

8.1.5 Record keys

CODASYL DBMSs allow to retrieve records based on access keys. Those access keys have to be declared in the DDL code as *record keys*. A declared record key has a name and specify the set of record fields used as access key.

8.1.6 Currency indicators

Since CODASYL DML commands manipulate data on a *one-record-at-a-time* basis, it is necessary to identify specific records in the database as *current records*. CODASYL DBMSs keep track (in the UWA) of the most recently accessed records and set occurences by means of a mechanism called *currency indicators*. Those indicators represent static predefined cursors that form the basis for the navigational facilities across the data. They include:

- ***Current of run unit***: the current record of the run unit is the last record involved in a successful query;
- ***Current of record type***: a current record is associated to each record type; it corresponds to the last accessed record of that type;
- ***Current of set type***: for each set type, there is a current record representing the last record accessed in the set (this can be either the owner or a member of the current set);
- ***Current of area***: the current record of an area refers to the last record accessed in the area;
- ***Current of record key***: the current record of a record key is the the last record that was accessed through that particular key type.

8.1.7 Status indicators

The UWA also comprises registers that inform the program on the status of the DML operations. The execution of any database manipulation statement causes the DBCS to insert a value into the special database status register (e.g., `DB-STATUS`). If no exceptions are encountered during the execution of the statement, the status indicator is set to zero. If an exception is encountered, it is set to the appropriate value depending on the statement and the exception type.

8.1.8 DML statements

The main CODASYL DML statements are the following:

- The `READY` statement makes the contents of an area available for processing.
- The `FIND` statement locates a record in the database subject to a variety of selection expression options. This statement aims to establish a specific record occurrence in the database as the object of subsequent statements.
- The `GET` statement obtains (a part of) the contents of a located database record and makes them available to the program through its UWA. The object of the `GET` is the current record of the run unit.
- The `MODIFY` statement alters the contents of one or more data items in a record and/or changes its set relationships. The object of the `MODIFY` statement is the current record of the run unit.
- The `ERASE` statement removes one record from the data base, as well as all subordinate member records, if any. The current record of the run unit is the object of the statement.
- The `STORE` statement causes a new record to be stored in the database. This statement establishes the current record of the run unit.
- The `CONNECT` causes, under certain conditions, a record stored in the database to become a member of a particular set. The current record of the run unit is the object of the statement.
- The `DISCONNECT` removes, under certain conditions, a record from a specified set in which it resides as a member. The current record of the run unit is the object of the statement.
- The `FINISH` statement makes an area unavailable for further accesses.

8.2 Migration methodology

The general migration methodology discussed in this chapter is made of two main phases, namely system reverse engineering and system conversion. The underlying approach combines analysis, transformational and generative techniques. With respect to the reference model presented in Chapter 4, the adopted migration strategy is <D2,P1>: the database conversion process relies on an initial reverse engineering phase (D1) and the program conversion process makes use of *wrappers* (P2).

8.2.1 System reverse engineering

In most cases, there is no complete and up-to-date documentation of the legacy system, and in particular of the database. Therefore, the initial phase of our methodology consists in recovering sufficient information about the legacy system. This knowledge will then serve as a basis for the migration process itself.

Inventory The reverse engineering phase starts with an inventory process which has two main objectives. First, it checks that the system is made of a complete and consistent set of source code files. Second, it allows to get a rapid overview of the application architecture in order to evaluate the complexity of the migration task (van Deursen and Kuipers, 1998), as well as the part of the work that cannot be automated. The results of the inventory step include the call graph, the database usage graph together with detailed statistics about the legacy database instructions (format, location, record types accessed).

Database reverse engineering The second main step concerns the reverse engineering of the database. This step aims at recovering the *conceptual schema* that expresses the exact information structure and meaning of the source database. Our database reverse engineering methodology (Hainaut, 2002) starts with the DDL (Data Definition Language) analysis process, parsing the legacy DDL code to retrieve the source *physical schema*. The schema refinement step (Henrard, 2003) consists of an in-depth inspection of the way the application programs use and manage the data, in order to produce the legacy *logical schema*. Through this process, additional structures and constraints are identified, such as foreign keys or finer-grained data structures, which are not explicitly declared in the DDL code but coped with in the procedural code. The legacy data may also be analyzed, either to detect constraints, or to confirm or discard hypotheses on the existence of such constraints. The final step is the data structure conceptualization, that interprets the legacy logical schema into the *conceptual schema*. The logical and the conceptual schemas have the same semantics, i.e., they cover the same informational content, but the latter is platform-independent.

8.2.2 System conversion

Database schema conversion The database schema conversion step translates the results obtained during the database reverse engineering process, that is, the conceptual schema of the database, into an equivalent database structure expressed in the target technology. The schema conversion process is driven by standard database design techniques, and can be modeled by a chain of semantics-preserving schema transformations (Hainaut, 2006). It is important to observe that this schema reengineering approach allows to obtain a *native*, fully-normalised relational schema, rid of the technical constructs of the legacy database.

Data migration Once the legacy schema has been converted, the data instances can be migrated towards the target platform. This is the goal of the data migration step, which involves data transformations deriving from the way the legacy schema was transformed into the target schema.

Program conversion In the context of database migration, program conversion is the modification of the programs so that they now access the renovated database instead of the legacy data. The functionalities of the program are left unchanged, as well as its programming language and its user interface (they can migrate independently). High quality program conversion generally is a complex process in that it relies on the rules used to transform the legacy schema into the target schema. In this chapter, we particularly elaborate on a wrapper-based approach to program conversion. While the discussion is based on the particular case of the conversion of a CODASYL database into a relational database, the approach is, to a large extent, valid to support other migration scenarios.

8.3 Wrapper-based program conversion

When the data management system (DMS) and the structure of the database change, the programs (more specifically the database manipulation statements) must be changed accordingly. Our strategy attempts to keep the legacy program logic unchanged and to map it on the new DMS technology, by means of *data wrappers*. A data wrapper is a *data model conversion* component that is called by the application program to carry out operations on the database. Its goal is generally to provide application programs with a modern interface to a legacy database (e.g., allowing Java programs to access COBOL files). In the migration context, the wrapper is actually a *backward* wrapper (Thiran et al., 2006), which allows the renovated data to be transformed into the legacy format in order to allow programs to read (`FIND/GET`), modify (`MODIFY`), write (`STORE/CONNECT`) and delete (`ERASE`) records that are built from rows extracted from a relational database. In this way, the application programs invoke the wrapper instead of the legacy DMS. If the wrapper simulates the modeling paradigm of the legacy database and its interface, the alteration of the legacy code is minimal. It mainly consists in replacing the data manipulation statements with wrapper invocations. The wrapper converts all legacy DMS requests from legacy applications into requests against the new DMS that now manages the data. Conversely, it captures results from the new DMS, converts them to the appropriate legacy format (Papakonstantinou et al., 1995) and delivers them to the application program. According to our approach, depicted in Figure 8.1, we develop one wrapper per legacy migrated record type. The wrappers may need to call each other in order to correctly simulate some legacy data manipulation primitives.

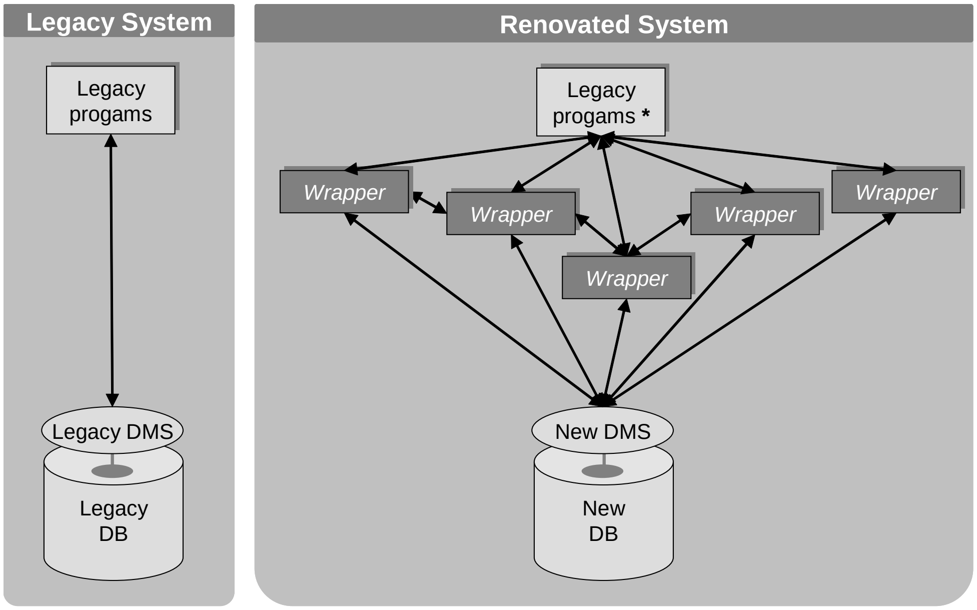


Figure 8.1: Migration architecture.

8.3.1 Schema mapping

The starting point of our wrapper-based methodology is the mapping between the source and target databases. In practice, such a mapping often necessitates three schemas:

- **The source physical schema** is the structure of the legacy database as defined in the DDL code. This schema contains the names of the different objects as they are used to navigate through the database, including access keys, identifiers and sorting keys.
- **The source logical schema** is the schema that programmers need to know in order to write or modify programs manipulating the database. The logical schema improves the physical schema by including implicit constructs such as finer-grained record structure, unique keys and foreign keys.
- **The target physical schema** is the structure of the migrated database.

The reason why both source and target physical schemas are necessary is easy to understand. The legacy physical schema is used to analyse the queries in the legacy code and the new physical schema serves as a basis for writing equivalent queries on top of the new database. However, these two schemas are not sufficient when the gap between them is so important that some information is missing when

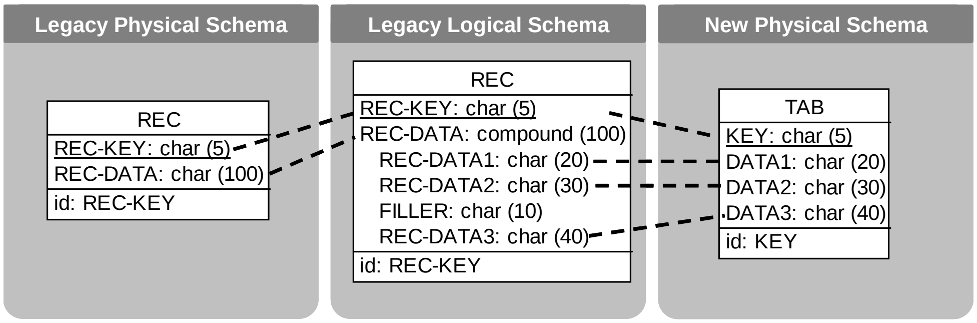


Figure 8.2: Mapping between legacy physical record, legacy logical record and new relational table.

transforming the data instances from one format to the other. For instance, the legacy physical schema may contain large fields that are transformed into several columns in the target physical schema. The intermediate logical schema contains the exact decomposition of the legacy fields, allowing to properly link the source and target physical schemas. Such a case is illustrated in Figure 8.2, where `REC-DATA` is decomposed (in the logical schema) into four sub-fields among which one is a *filler*. This is a usual practice in legacy systems: the filler is used as a reserved space *for future use*. This is a way to avoid the modification of the structure of a record, which may be a complex and time-consuming task.

8.3.2 Wrapper development

The wrappers are the central part of the program conversion phase. They simulate the legacy DML [2], such that the logic of the legacy programs need not be adapted. As mentioned above, we use one distinct wrapper for each legacy record type that is migrated. Each wrapper implements all the actions that can be performed on its associated record type. To do so, it may call other wrappers when necessary. The responsability of the wrappers is twofold:

- **structure conversion**: the wrappers adapt the legacy database queries to the target structure;
- **language conversion**: the wrappers translate the legacy DML primitives using target DML commands;

8.3.2.1 Structure conversion

The first and most obvious function of the wrapper is to convert data structures. It reads the data from the relational database and presents them to the programs

[2] Data Manipulation Language

according to the legacy format. Conversely, it receives data from the programs in the legacy format and stores them in the relational database. This conversion is based on the mapping that holds between the source and target database schemas. Here below, we discuss three typical issues occuring when mapping CODASYL data structures to the relational model.

Record types and fields In the easiest mapping pattern, called one-to-one mapping, each legacy record type is mapped to an SQL table and each field is mapped to a column. This scenario mainly requires data type conversion.

The second typical mapping pattern captures the situation when one record is mapped with one table, but some fields are translated into several columns (or conversely), and some (parts of the) fields may not be migrated. To cope with this mapping pattern, we use the logical schema. Each field of the physical schema is mapped with one (compound) field of the logical schema, and each column derives from one field of the logical schema (Figure 8.2). Now, the logical schema is in a one-to-one relationship with the relational structures, since each of its fields corresponds to one column at most.

The third mapping pattern occurs when one legacy field is converted into an additional table. This transformation is applied when the field is multivalued, i.e., when it represents an array or a list of values. The mapping is used to identify the SQL table that translates the field in order to generate queries retrieving/updating rows from/of the table. The table representing the field must be connected to the table translating its record type through a foreign key. The last two mapping patterns can also be combined.

Set types Each CODASYL set type is translated into an SQL foreign key from the *member table* to the *owner table*. According to the nature of the set type, the corresponding foreign key columns are declared nullable or not. Using this way of translating set types, CODASYL conditions related to sets are easy to simulate. For instance, the conditions of the form `IF` *setName* `MEMBER` that verify that the current record is a member of a given set, can be simulated by checking that the foreign key associated to *setName* is not null for the corresponding row. As another example, the *setName* `IS NOT EMPTY` condition, verifying that the current record of a given set has member records connected to it, can be translated by counting the rows of the member table that reference the current owner through the corresponding foreign key.

DB-KEY In CODASYL the special register `DB-KEY` can be used to identify a record in the database. This technical reference is global to the database (i.e., unique for all records whatever their type). This functionality is not available in SQL. Some RDBMS provide a *row-id* that is local to a table only. A possible solution consists in adding a *db-key column* to each SQL table translating a legacy record type. This column identifies a row in the table and is usually implemented by a *auto-num* data type. The DB-KEY concept is then represented by a compound variable that contains the name of the record type together with the value of the db-key column. This compound value is thus unique in the database.

8.3.2.2 Language conversion

Simulating the CODASYL language on top of a relational database poses non trivial challenges such as the following:

- **currency indicators management**: the wrappers are statefull in that they must store, manage and synchronize the CODASYL currency indicators, which represent current states of reading sequences;
- **sequence management**: the wrappers simulate the CODASYL reading and writing sequences on top of the target relational database;
- **error management**: the wrappers detect and report errors back to the legacy programs. The errors are detected on top of the relational database, but are reported by means of CODASYL status indicators;

Currency indicators management A wrapper associated to a record type stores and manages (1) the current record of the record type, (2) the current records of the set types for which the record type is a member, and (3) the current record of the key belonging to the record type. Each currency indicator comprises (1) a flag, indicating the current record status (null, not null, erased[3]), and (2) the value of the current record itself.

The major difficulty when simulating CODASYL primitives in SQL is to properly position the currency indicators in case of crosscutting reading sequences. Let us illustrate this issue by a small example. The top-left part of Figure 8.3 depicts a fragment of a CODASYL database schema, where record type `ORDERS` (member) is connected to record type `CUSTOMER` (owner) through set type `S`. The bottom-left part of the figure represents sample data instances of that schema (`C1` and `C2` are of type `CUSTOMER` and `O1` , ..., `O5` are of type `ORDERS`). The top-right side of the same figure represents the corresponding relational schema fragment, where (1) table `CUSTOMER` translates record type `CUSTOMER`, (2) table `ORDERS`translates record type `ORDERS`, and (3) set type `S` is translated into a foreign key column `ORDERS.Owner` targeting `CUSTOMER.ID`. The bottom-right part of the figure depicts how the data instances are organized in the relational database.

Let us now assume that, at some point of the program execution, the current record of set type `S` is `C1`. From that point, Figure 8.4 considers a sequence of `FIND` statements, together with their impact on the current record of `S`.

The record identified by statements of the form `FIND NEXT WITHIN S` is the next record in the set, relative to the current record of the set type `S` according to the ordering criteria specified for set type `S`. We notice that when record `O4` is directly accessed through its `DB-KEY` , the program switches to a different set occurence since

[3] In CODASYL, when a record is erased, its ghost keeps being the current record!

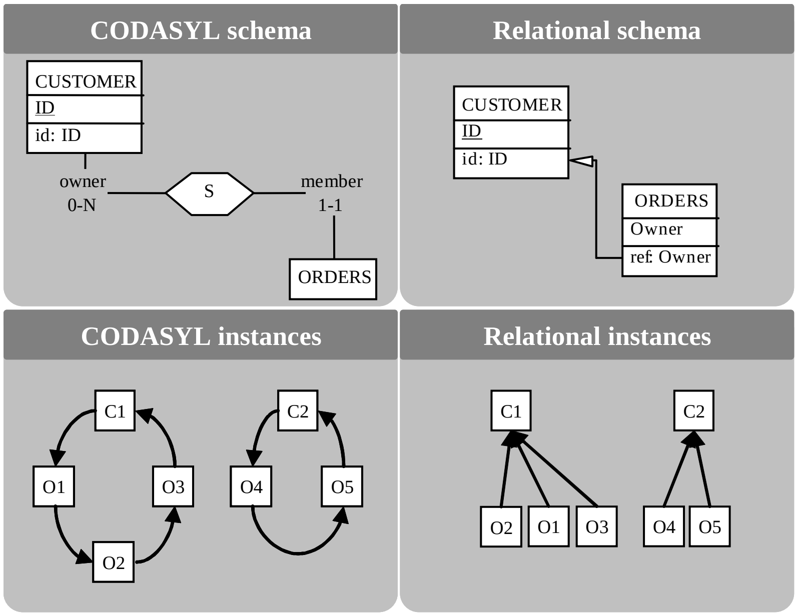


Figure 8.3: A CODASYL schema fragment and its corresponding relational schema, together with sample data instances.

Executed `FIND` statement	**Current record of `S`**
...	`C1`
`FIND NEXT WITHIN S`	`O1`
`FIND NEXT WITHIN S`	`O2`
`FIND ORDERS DB-KEY <db-key(O4)>`	`O4`
`FIND NEXT WITHIN S`	`O5`

Figure 8.4: Impact of successive `FIND` statements on the current record of a set type

Set current record	Set cursor state	Set cursor action(s)
member	open	fetch
member	closed	re-position to current, fetch
owner	closed	open, fetch

Table 8.1: SQL translation of `FIND NEXT WITHIN S` statements.

`O4` has another owner than `O2`. From that point, the next record of the set is not `O3` anymore, but is now `O5`.

Our approach for simulating such a behaviour in SQL is as follows. For each set type `S` , a dedicated SQL cursor is used for traversing the set occurences of `S` . In our example of Figure 8.3, the SQL cursor would be of the following form:

```
DECLARE S CURSOR FOR
SELECT <columns>
FROM ORDERS
WHERE Owner = <current_owner_id>
ORDER BY <set_ordering_criteria>
```

This cursor selects the rows representing all the members of the current set. Indeed, all these rows refer to the owner of the current set through the associated foreign key (`Owner`). The rows belonging to the set cursor are ordered in accordance with the specified set ordering criteria. Each set cursor is used within the wrapper associated to the member record (`ORDERS` in our example).

Table 8.1 summarizes the SQL translation of statements of the form `FIND NEXT WITHIN S`. Most of the time, the wrapper only needs to execute a `fetch S` command, provided that cursor `S` is already open. The wrapper closes the set cursor each time the program switches to a new set. This can happen either when the current member changes (from `O2` to `O4` in our example), or when the current owner is modified. In the second case, the wrapper of the owner calls the wrapper of the member, and the latter closes the corresponding set cursor.

Re-positioning the reading sequence to the current record of a set type `S` can be done by re-opening cursor `S` and fetching it until the current record of `S` is reached. An alternative solution consists in opening another cursor `S'`, similar to cursor `S`, which selects all the *remaining* members of the current set starting from the current record of `S`. The latter solution is preferable for performance reasons.

Note that the wrappers must deal with another complication: `FIND` primitives provide an option (called *retaining phrase*) that allows programmers to select currency indicators to be left unchanged when executing the primitive.

Sequence management An important difference between CODASYL and SQL concerns the management of reading and writing sequences. In CODASYL, when records are retrieved from the database, they are read according to the physical order, no sorting is done at the time of database reading. Consequently, the order in which the elements are written in the database is very important and is specified in the DDL code. For example, `INSERTION NEXT` (resp. `LAST`) means that a new record is inserted right after the current record (resp. as the last element). A sorting key may also be specified, which causes the records to be stored according to the corresponding order.

By contrast, in SQL the physical order of the data is immaterial, the order in which the rows appear being explicitly specified in the *order by* clause. Therefore, in order to retrieve the SQL rows in the same order as the CODASYL records, the sorting criteria must be made explicit. If a sorting key is declared in the DDL code, translating this condition in SQL is straightforward. But in case no explicit sorting key was specified, two main situations can arise. First, the programs may use an *implicit* sorting key that was not declared in the DDL code (typically for performance reasons). In this case, this implicit key can be used, provided that it is recovered. The second possible situation is when the programmer does not expect the data to be read in a particular order. In this situation, we use the table identifier as sorting key. Indeed, due to currency indicators management issues (e.g., cursor re-positioning) we have to make sure that the order be always the same, which is not guaranteed by SQL.

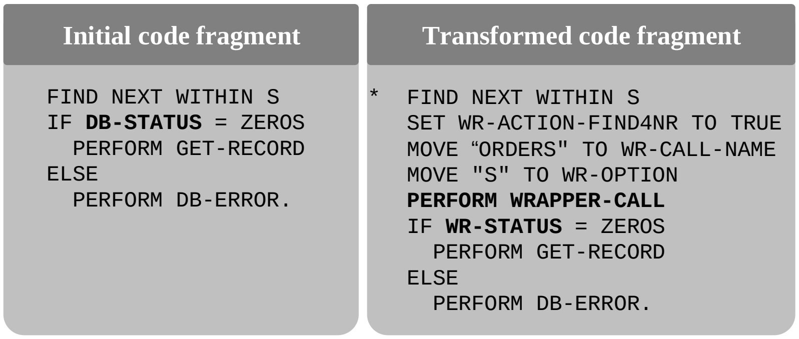
Error management The execution of any CODASYL statement causes the data management system to insert a value into the special register `DB-STATUS`. This value is made up of a statement code in the leftmost two character positions, and a status code in the rightmost five character positions. If the execution of the data manipulation statement results in an exception condition, the statement code indicates which type of database manipulation statement caused the exception condition (e.g., 05 for `FIND statements`), and the status code indicates which type of database exception condition occurred. If the statement execution does not result in an exception, the indicator `DB-STATUS` is set to zero.

The wrappers must reproduce this behaviour when accessing the relational database. As an example, let us assume that the current record of set type `S` is `O3`. If `FIND NEXT WITHIN S` is executed, the end of the set is reached. In CODASYL, such an exception condition is indicated by a status code value equal to 02100. Thus the invoked wrapper should return an exception status value equal to 0502100. In terms of the SQL language, this corresponds to the end of the set cursor, which is indicated by a `SQLCODE` value equal to 100.

8.3.3 Program transformation

During the program transformation step, the legacy application programs are interfaced with the wrappers. The source code adaptation involves the combination of the following transformations:

- replacing legacy database primitives with corresponding wrapper invocations (*db-2-wrap*);
- renaming some variables (*var-rename*);



Initial code fragment

```
FIND NEXT WITHIN S
IF DB-STATUS = ZEROS
  PERFORM GET-RECORD
ELSE
  PERFORM DB-ERROR.
```

Transformed code fragment

```
*  FIND NEXT WITHIN S
   SET WR-ACTION-FIND4NR TO TRUE
   MOVE “ORDERS" TO WR-CALL-NAME
   MOVE "S" TO WR-OPTION
   PERFORM WRAPPER-CALL
   IF WR-STATUS = ZEROS
     PERFORM GET-RECORD
   ELSE
     PERFORM DB-ERROR.
```

Figure 8.5: Example of legacy code transformation.

- adapting the type of some variables (*type-adapt*);
- adding new variable declarations (*var-insert*);
- adding new code sections (*code-insert*).

The exact combination chosen depends on the kind of source code file to be transformed, as identified during the reverse engineering phase. For instance, the *db-2-wrap* transformation is useless when converting programs that do not directly access the legacy database.

The main input parameters of the program transformation tools are automatically derived from the source, intermediate and target database schemas. These parameters include (1) the list of the record types; (2) the list of the *migrated* record types; (3) the list of the area's; (4) the correspondence table between set types, members and owners; (5) additional variable declarations and code fragments.

Figure 8.5 illustrates the program transformation step. It shows how `FIND NEXT WITHIN S` statements are rewritten as corresponding wrapper invocations (*db-2-wrap*). In this particular case, the transformed program specifies the action to perform (`SET WR-ACTION-FIND4NR TO TRUE`), as well as the set type involved (`MOVE "S" TO WR-OPTION`). The wrapper to invoke is the one associated to the member of the given set type `S`, namely `ORDERS` (`MOVE "ORDERS" TO WR-CALL-NAME`). Figure 8.5 also gives an example of variable renaming (*var-rename*): CODASYL status indicator `DB-STATUS` is renamed as wrapper status indicator (`WR-STATUS`). The wrappers use the renamed variable to report status indicators to the application programs.

8.4 CODASYL-to-SQL translation

In this section, we systematically present conversion rules for simulating the main CODASYL data manipulation primitives on top of the SQL language. According to the approach presented in this chapter, those rules serve as a basis for the automatic generation of backward wrappers. The variant of the CODASYL language considered is IDS/II, the implementation of CODASYL used on Bull GCOS mainframes.

8.4.1 Preliminaries

One-to-one mapping As for the COBOL-to-SQL rules of Chapter 7, we will assume a one-to-one mapping holding between the CODASYL schema and the relational schema. This mapping can be expressed as follows. Let $\mathcal{S}_{cod}$ be the CODASYL schema. Let $\mathcal{S}_{rel}$ be its relational translation. We assume that:

- $\forall$ record type $R \in \mathcal{S}_{cod}$: $\exists$ a corresponding table $t_R \in \mathcal{S}_{rel}$;
- $\forall$ record type $R \in \mathcal{S}_{cod}$: $\forall$ field f_i of R : $\exists$ a corresponding column $c_{f_i} \in t_R$;
- $\forall$ set type $S \in \mathcal{S}_{cod}$ between a member record type *mem* and an owner record type *own* : $\exists$ a foreign key fk_S from t_{mem} to t_{own}. We will note id_S the identifier of table t_{own} corresponding to foreign key fk_S in t_{mem}.

Currency and status registers We assume the program fragments (wrappers or additional procedures) in charge of simulating the IDS/II primitives in SQL make use of the following variables:

- *cur* denotes the current record of the run unit;
- $\forall$ record type $R \in \mathcal{S}_{cod}$: `UWA-`R represents the record R of the user working area (UWA), i.e., at the program side;
- $\forall$ record type $R \in \mathcal{S}_{cod}$: `WK-`R represents a *working* variable for storing a record of type R;
- $\forall$ record type $R \in \mathcal{S}_{cod}$: `CUR-`R represents the *current record of* R;
- $\forall$ set type $S \in \mathcal{S}_{cod}$: `CUR-`S represents the *current record of* S;
- $\forall$ access key type $K \in \mathcal{S}_{cod}$: `CUR-`K represents the *current record of* K;
- $\forall$ set record type $R \in \mathcal{S}_{cod}$: `INIT-FLAG-`R is a flag indicating whether the *current record of* R is null (= 0) or initialized (= 1).
- $\forall$ set type $S \in \mathcal{S}_{cod}$: `INIT-FLAG-`S is a flag the value of which indicates whether the *current record of* S:
  - is null (= 0);
  - is a member record and the set cursor is well-positioned (= 1);
  - is the owner record and the set cursor is closed (= 2);
  - was deleted (= 3);

  - is a member record but the set cursor is not well-positioned (= 4);
  - is the owner record and the set cursor is open (= 5).

- `STATUS` is the status variable indicating the IDS/II exceptions, it simulates the IDS/II register `DB-STATUS`;

- ∀ cursor c : `LC-`c is a boolean variable that indicates whether c is the last opened cursor or not.

Help COBOL paragraphs The translation rules below will make use of the following COBOL procedures:

- `MOVE-UWA-TO-WK-`R: moves the program-side record `UWA-`R from the UWA to the working variable `WK-`R.

- `MOVE-WK-`R`-TO-CUR`: moves the working variable `WK-`R to *cur*, to `CUR-`R, and to `CUR-`S (∀ set type S : R is member/owner of S), according to the *retaining phrase* if any.

- `MOVE-WK-`R`-TO-CUR-`K: moves the working record `WK-`R to the current record of key type K (`CUR-`K).

- `MOVE-CUR-TO-UWA-`R: delivers the current record of the run unit, which is of type R, to the user working area of the program (i.e., to `UWA-`R).

- `CLOSE-LAST-CURSOR-`$recName$: closes the last opened database cursor used to access record type $recName$.

- `CLOSE-LAST-CURSOR-`$keyName$: closes the last opened cursor based on record key type $keyName$.

- `UPDATE-STATUS`: updates the value of the status variable (`STATUS`) and of the `INIT-FLAG-`x variables, if necessary.

8.4.2 `FIND` statement

This section is dedicated to the simulation of `FIND` statements in SQL.

General format

Syntax `FIND` $recordSelectionExpression$
`[RETAINING CURRENCY FOR { REALM |SETS |RECORD |{`$setName_1$ `[,`$setName_2$`] ...}]`

Effect The FIND statement causes the record referenced by the record selection expression to become the current record of the run unit.

If the RETAINING phrase is not specified, the referenced record also becomes the current record of its area, the current record of its record type, and the current record of all sets in which it is a tenant (owner or member).

If the RETAINING phrase is specified with:

- REALM: the area currency indicators are left unchanged;
- SETS: no set type currency indicators are changed;
- RECORD: the record type currency indicators remains unchanged;
- $setName_1, setName_2, ...$: the set currency indicators for the named sets are not changed.

Below, we describe how the various kinds of FIND statements can be simulated by means of corresponding select queries or SQL cursor manipulation statements[4].

Format 1

Syntax FIND [*recName*] DB-KEY IS *identifier*

Effect This statement identifies the record of type *recName* whose database key value is equal to the value of the data item referenced by *identifier*. If such a record is found, it becomes the *current record*. If a record with the specified database key value is not found, an exception status (0502400) is returned and the currency indicators are left unchanged.

SQL translation Translating this statement in SQL can be done provided that the database key has been explicitly defined as a column (e.g., dbkey) of the relational table translating *recName* ($t_{recName}$). If it is the case, the SQL translation consists in selecting the corresponding row as follows:

```
FIND1-$recName$.
  MOVE ZERO TO STATUS.
  MOVE $identifier$ TO DBK.
  EXEC SQL
     SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
     INTO :WK-$recName$-dbkey, :WK-$recName$-$f_1$, ... :WK-$recName$-$f_n$
     FROM $t_{recName}$
     WHERE dbkey = :DBK
  END-EXEC.
  IF SQLCODE NOT = ZERO
```

[4] For readability reasons, we will ignore some exception cases in the provided translation rules. Those cases are taken into account by our wrapper generator.

```
    MOVE 0502400 TO STATUS
  ELSE
    PERFORM MOVE-WK-$recName$-TO-CUR
  END-IF.
END-FIND1-$recName$.
  PERFORM UPDATE-STATUS.
```

Format 2

Syntax `FIND {ANY |DUPLICATE }` *recName*

Rules The location mode corresponding to record type *recName* must be *calculation.*

Variants We distinguish two possible instructions from the syntax:

- `FIND ANY` *recName* (FIND2A);
- `FIND DUPLICATE` *recName* (FIND2D).

Effect

- **FIND2A**: If the `ANY` phrase is specified, the *calculation key values* for the record named by *recName* (i.e., its identifying field values) are used to locate the record in the database. The values must be moved into the corresponding fields of the UWA before the `FIND` statement is issued. If more than one record of type *recName* contains calculation key values equal to those in the record area, the record identified is the one whose database key is the lowest.
- **FIND2D**: If the `DUPLICATE` phrase is specified, the first record found after the current record of the run unit, with the proper calculation key values is the record identified. If a record with the proper calculation key values is not found an exception status (0502400) is returned and the currency indicators are left unchanged.

SQL translation The SQL translation of both instructions makes use of the *calc key cursor* defined as follows:

```
EXEC SQL
  DECLARE CK_$recName$ CURSOR FOR
  SELECT dbkey, $c_{f_1}, \ldots\ c_{f_n}$
  FROM $t_{recName}$
  WHERE $c_{f_{1_{ck}}}$ = :WK-$recName$-$f_{1_{ck}}$ AND
        ...
        $c_{f_{p_{ck}}}$ = :WK-$recName$-$f_{p_{ck}}$
  ORDER BY dbkey
END-EXEC.
```

where the calculation key of *recName* is computed from fields $f_{1_{ck}} \dots f_{p_{ck}}$. The calc key cursor selects the rows of the table translating *recName* having the proper value(s) of the column(s) $c_{i_{ck}}$ corresponding to fields $f_{i_{ck}}$.

- **FIND2A.** The `FIND ANY` *recName* primitives may then be simulated as follows:

```
FIND2A-recName.
  MOVE ZERO TO STATUS.
  PERFORM CLOSE-LAST-CURSOR-recName.
  MOVE UWA-recName-$f_{1_{ck}}$ TO WK-recName-$f_{1_{ck}}$.
  ...
  MOVE UWA-recName-$f_{p_{ck}}$ TO WK-recName-$f_{p_{ck}}$.
  EXEC SQL
    OPEN CK_recName
  END-EXEC.
  IF SQLCODE NOT = ZERO
    GO TO END-FIND2A-recName
  END-IF.
  EXEC SQL
    FETCH CK_recName
    INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
    MOVE 0502400 TO STATUS
  ELSE
    PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND2A-recName.
  PERFORM UPDATE-STATUS.
```

  The translation consists in opening the calc key cursor, before fetching it. If the record is found, then it becomes the current record. By contrast, if the cursor is empty (end of cursor), the exception status (0502400) is returned.

- **FIND2D.** The `FIND DUPLICATE` *recName* primitives may be simulated as follows:

```
FIND2D-recName.
  MOVE ZERO TO STATUS.
  IF NOT(LC-CK_recName)
    PERFORM FIND2A
  END-IF.
  EXEC SQL
    FETCH CK_recName
    INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
```

```
    MOVE 0502400 TO STATUS
  ELSE
    PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND2D-recName.
  PERFORM UPDATE-STATUS.
```

In case the last opened cursor is not the calc key cursor, the FIND2D procedure simply calls the FIND2A procedure before fetching again the cursor. If the cursor is already open, the procedure simply fetches it. If the record is found, then it becomes the current record. By contrast, if there is no remaining row (end of cursor), an exception status code (0502400) is returned.

Format 3

Syntax `FIND DUPLICATE WITHIN` *setName* `USING` $field_1$ `[,`$field_2$`] ...`

Effect The record identified (1) is a member of the set occurrence identified by the currency indicator for *setName*, and (2) has the contents of the data items referenced by $field_1$, $field_2$,... equal to those in the current record of *setName*. The search begins at the next record in the order defined by the set ordering criteria of *setName*.

SQL translation Translating this primitive, the code name of which is FIND3, necessitates another cursor which selects all the records that (1) are a member of the current set occurence of *setName*, and (2) are located *after* the current record of the set according to the set ordering criteria. This cursor, that we call *set after current cursor*, is defined as follows:

```
EXEC SQL
   DECLARE FK_setName_AC CURSOR FOR
   SELECT dbkey, c_f1, ... c_fn
   FROM t_recName
   WHERE fk_setName = :CUR-setName-fk_setName AND
         orderingCriteria_setName > :CUR-setName-orderingCriteria_setName
   ORDER BY orderingCriteria_setName
END-EXEC.
```

where *recName* is the member record type of set type *setName*, and $orderingCriteria_{setName}$ denotes the field(s)/column(s) involved in the set ordering criteria of *setName*.

Simulating the FIND3 primitive basically consists in opening the `FK_`*setName*`_AC` cursor (if it is not already open), before fetching it until a record with the proper values of $field_1$ `[,`$field_2$`]`... is found. If such a record cannot be found, the exception status code 0502400 (*no record found to satisfy record selection criteria*) is returned. We give below the pseudo-code of this translation:

```
FIND3-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0
    MOVE 0503100 TO STATUS
    GO TO END-FIND3-setName
  END-IF.
  IF INIT-FLAG-setName = 2 OR 3 OR 5
    MOVE 0503800 TO STATUS
    GO TO END-FIND3-setName
  END-IF.
  IF NOT(LC-FK_setName_AC)
    EXEC SQL
       OPEN FK_setName_AC
    END-EXEC
  END-IF.
  IF SQLCODE NOT = ZERO
    GO TO END-FIND3-setName
  END-IF.
  MOVE 0 TO FOUND.
  PERFORM
    UNTIL FOUND = 1 OR SQLCODE NOT = ZERO
    EXEC SQL
       FETCH FK_setName_AC
       INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
    END-EXEC
    IF SQLCODE NOT = ZERO
       GO TO END-FIND3-setName
    END-IF
    IF WK-recName-$field_1$ = CUR-recName-$field_1$ AND
        WK-recName-$field_2$ = CUR-recName-$field_2$ AND
        ...
       MOVE 1 TO FOUND
    END-IF
  END-PERFORM.
  IF SQLCODE NOT = ZERO
    MOVE 0502400 TO STATUS
    GO TO END-FIND3-setName
  ELSE
    PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND3-setName.
  PERFORM UPDATE-STATUS.
```

where *recName* is the member record type of set type *setName*.

The exception cases are the following. If the current record of set type *setName* is not initialized, the exception status 0503100 (*current record is null*) is returned.

If the current record of *setName* is owner of the set (`INIT-FLAG-`*setName* `= 2 OR 5`) or was deleted (`IF INIT-FLAG-`*setName* `= 3`), the exception status 0503800 (*current record set is not correct record type*) is returned.

Format 4

Syntax `FIND {FIRST |NEXT |LAST |PRIOR}` [*recName*] `WITHIN` {*setName* |*areaName*}

Variants We can distinguish two main categories of FIND4 statements:

- `FIND` ... `WITHIN` *setName* statements, that navigate in the current set occurence of *setName*;
- `FIND` ... `WITHIN` *areaName* statements, which target the records belonging to a specified area *areaName*.

We will ignore here the translation of the second category of statements since we assume the areas are not translated in the target relational database. Indeed, such physical collections of records do not usually make sense in the relational model. Actually, we identify two typical cases. In the most frequent case, the areas are simply ignored when migrating the database. In this case, the "`WITHIN` *areaName*" clause can be ignored as well in the SQL translation, which then considers *all* the records of type *recName*. The second possible situation is when the areas actually classify the records according to *logical* criteria. In this second case, one must specify which (additional) columns of the corresponding tables allow to determine to which area a record belong, such that an appropriate cursor can be defined. Note that in both cases, the SQL translation is possible if and only if *recName* is specified.

We will therefore only present the translation of the first category of FIND4 statements, which can be further subdivided in eight variants.

Effect

- **FIND4FR**: `FIND FIRST` *recName* `WITHIN` *setName*.
  The record identified is the *first member* of the set in which the current record of the set type referenced by *setName* is a tenant (owner or member), according to the set ordering criteria for that set type.
- **FIND4NR**: `FIND NEXT` *recName* `WITHIN` *setName*.
  The record identified is the *next* record in the set, relative to the current record of the set type referenced by *setName* according to the set ordering criteria for that set type.
- **FIND4LR**: `FIND LAST` *recName* `WITHIN` *setName*.
  The record identified is the *last member* of the set in which the current record of the set type referenced by *setName* is a member, according to the set ordering criteria for that set type.

- **FIND4PR**: `FIND PRIOR` *recName* `WITHIN` *setName*.
  The record identified is the *prior* record in the set, relative to the current record of the set type referenced by *setName* according to the set ordering criteria for that set type.

- **FIND4F**: `FIND FIRST WITHIN` *setName*.
  This variant is equivalent to FIND4FR, where *recName* is the member record type of *setName*.

- **FIND4N**: `FIND NEXT WITHIN` *setName*.
  This variant is equivalent to FIND4NR, where *recName* is the member record type of *setName*.

- **FIND4L**: `FIND LAST WITHIN` *setName*.
  This variant is equivalent to FIND4LR, where *recName* is the member record type of *setName*.

- **FIND4P**: `FIND PRIOR WITHIN` *setName*.
  This variant is equivalent to FIND4PR, where *recName* is the member record type of *setName*.

SQL translation

- **FIND4FR.** The SQL translation of the FIND4FR primitive makes use of the following cursor.

```
EXEC SQL
   DECLARE FK_$setName$ CURSOR FOR
   SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
   FROM $t_{recName}$
   WHERE $fk_{setName}$ = :CUR-$setName$-$fk_{setName}$
   ORDER BY $orderingCriteria_{setName}$
END-EXEC.
```

  This cursor selects all the member records of the set occurence of *setName* in which the current record of *setName* is a member. The records are ordered according to the set ordering criteria associated to *setName*.

  Translating the FIND4FR primitive simply consists in opening and fetching this cursor, as performed in the following pseudo-code fragment:

```
FIND4FR-$setName$.
  IF INIT-FLAG-$setName$ = 0
    MOVE 0503100 TO STATUS
    GO TO END-FIND4FR-$setName$
  END-IF.
  MOVE ZERO TO STATUS.
  PERFORM CLOSE-LAST-CURSOR-$recName$.
  EXEC SQL
    OPEN FK_setName
  END-EXEC.
  IF SQLCODE NOT = ZERO
    GO TO END-FIND4FR-setName
  END-IF.
  EXEC SQL
    FETCH FK_setName
    INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
    MOVE 0502100 TO STATUS
    PERFORM CLOSE-LAST-CURSOR-recName
    GO TO END-FIND4FR-setName
  ELSE
    PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND4FR-setName.
  PERFORM UPDATE-STATUS.
```

If the current record of the set type referenced by *setName* is not initialized, the exception status code 0503100 (*current record is null*) is returned.

If the set is empty, the exception status code 0502100 (*end of set*) is returned.

- **FIND4NR.** Translating FIND4NR primitives is a bit more complex, as already suggested in Section 8.3.2.2. The pseudo-code of the translation is provided below:

```
FIND4NR-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0
    MOVE 0503100 TO STATUS
    GO TO END-FIND4NR-setName
  END-IF.
  IF INIT-FLAG-setName = 2 OR 5
    PERFORM FIND4FR-setName
    GO TO END-FIND4NR-setName
  END-IF.
  IF NOT(LC-FK_setName) OR INIT-FLAG-setName = 4
    PERFORM OPEN-AND-POSITION-FK_setName
    IF SQLCODE NOT = ZERO
       MOVE 0502100 TO STATUS
       GO TO END-FIND4NR-setName
    END-IF
  END-IF.
  EXEC SQL
    FETCH FK_setName
    INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
    MOVE 0502100 TO STATUS
    PERFORM CLOSE-LAST-CURSOR-$recName$
    GO TO END-FIND4NR-$setName$
  ELSE
    PERFORM MOVE-WK-$recName$-TO-CUR
  END-IF.
END-FIND4NR-$setName$.
  PERFORM UPDATE-STATUS.
```

If the current record of set type *setName* is not initialized (`INIT-FLAG-`*setName* `= 0`), the exception status code 0503100 (*current record is null*) is returned.

If the current record of set type *setName* is the owner (`INIT-FLAG-`*setName* `= 2 OR 5`), the translation is (almost) the same as for the corresponding FIND4FR primitive. Indeed, the next record relative to the owner is the first member record of the set. The only difference is that one first has to move the value of the identifier of the current owner (`CUR-`$setName$-$id_{setName}$) to the foreign key value used as input of the set cursor (`CUR-`$setName$-$fk_{setName}$).

If the current record of the set type referenced by *setName* is a member record m, we must distinguish two main cases:

1. the set cursor is either closed (`NOT(LC-FK_`*setName*`)`) or not correctly positioned on m (`INIT-FLAG-`*setName* `= 4`): in this case, the translation must first re-open the set cursor and reposition it on the current record m[5]. Once this re-positioning has been done, an additional fetch of the set cursor is performed.
2. the set cursor is open and position on m: in this case, the translation simply consists in performing an additional fetch of the set cursor.

If the last fetch reaches the end of the set cursor, the exception status 0502100 (*end of set*) is returned.

- **FIND4LR.** The SQL translation of the FIND4LR uses the *reverse set cursor* defined as follows.

```
EXEC SQL
   DECLARE FK_$setName$_REVERSE CURSOR FOR
   SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
   FROM $t_{recName}$
   WHERE $fk_{setName}$ = :CUR-$setName$-$fk_{setName}$
   ORDER BY reverse($orderingCriteria_{setName}$)
END-EXEC.
```

[5] this is done through procedure `OPEN-AND-REPOSITION-FK_`*setName*.

where reverse($\mathit{orderingCriteria}_{setName}$) corresponds to the inverse ordering of $\mathit{orderingCriteria}_{setName}$.

The translation consists in opening and fetching the reverse set cursor, in order to retrieve the last member record of the current set occurence of *setName*.

```
FIND4LR-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0
    MOVE 0503100 TO STATUS
    GO TO END-FIND4LR-setName
  END-IF.
  PERFORM CLOSE-LAST-CURSOR-recName.
  EXEC SQL
    OPEN FK_setName_REVERSE
  END-EXEC.
  IF SQLCODE NOT = ZERO
    GO TO END-FIND4LR-setName
  END-IF.
  EXEC SQL
    FETCH FK_setName_REVERSE
    INTO :WK-recName-dbkey, :WK-recName-f1, ... :WK-recName-fn
  END-EXEC.
  IF SQLCODE NOT = ZERO
    MOVE 0502100 TO STATUS
    PERFORM CLOSE-LAST-CURSOR-recName
    GO TO END-FIND4LR-setName
  ELSE
    PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND4LR-setName.
  PERFORM UPDATE-STATUS.
```

If the current record of set type *setName* is not initialized, the exception status code 0503100 (*current record is null*) is returned.

- **FIND4PR.** The way of translating the FIND4PR primitives depends on the availability of the `FETCH PRIOR` statement in the target relational DBMS. If it is available, the set cursor can be fetched backwards. If it is not available, one can use either the *reverse set cursor*, which necessitates re-positioning, or the *reverse set before current cursor*, which selects all the member records of the current set occurence that are located *before* the current record, in the reverse order of the set ordering criteria. In both cases, the translation is very similar to the one of the FIND4NR primitive.

Format 5

Syntax `FIND CURRENT [`*recName*`] [ WITHIN {`*setName* |*areaName*`} ]`

Rules If *recName* and *areaName* are specified, then *recName* must be the record type of the current record of *areaName*. If *recName* and *setName* are specified, then *recName* must be the record type of the current record of *setName*.

Effect The effect of the FIND5 statement obviously depends on the variant used:

- **FIND5**: `FIND CURRENT`
  The record identified is the current record of the run unit.
- **FIND5R**: `FIND CURRENT` *recName*
  The record identified is the current record of the record type referenced by *recName*.
- **FIND5S** and **FIND5RS**: `FIND CURRENT [`*recName*`] WITHIN` *setName*
  The record identified is the current record of the set type referenced by *setName*.
- **FIND5A** and **FIND5RA**: `FIND CURRENT [`*recName*`] WITHIN` *areaName*
  The record identified is the current record of the area referenced by *areaName*.

For the same reasons as those explained above, we will ignore below the translation FIND5A and FIND5RA statements, since we assume the areas are not relevant, and therefore not translated, in the relational database.

SQL translation

- **FIND5**: the translation of the `FIND CURRENT` statement consists in making the current record of the run unit (*cur*) the current record of its record type and set types. In case *cur* is null, the exception status code 0503200 is returned (*current record of the run unit is null*).

```
FIND5.
  MOVE ZERO TO STATUS.
  IF cur = null
     MOVE 0503200 TO STATUS
     GO TO END-FIND5
  ELSE
     MOVE cur TO WK-recName
     PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND5.
  PERFORM UPDATE-STATUS.
```

  where *recName* refers to the record type of *cur*.

- **FIND5R**: the translation of the `FIND CURRENT` *recName* consists in making the current record of record type *recName* the current record the run unit and of its set types. If the current record of the specified record type *recName* is null, the exception status code 0503100 (*current record of record type is null*) is returned. This behaviour can be simulated as follows:

```
FIND5R-recName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-recName = 0
     MOVE 0503100 TO STATUS
     GO TO END-FIND5R-recName
  ELSE
     MOVE CUR-recName TO WK-recName
     PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND5R-recName.
  PERFORM UPDATE-STATUS.
```

- **FIND5S**: the translation of the `FIND CURRENT WITHIN` *setName* consists in making the current record of set type *setName* the current record the run unit and of its record type. If the current record of *setName* is null or was deleted, the exception status code 0503100 (*current record of set type is null*) is returned.

```
FIND5S-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0 OR 3
     MOVE 0503100 TO STATUS
     GO TO END-FIND5S-setName
  ELSE
     MOVE CUR-setName TO WK-recName
     PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND5S-setName.
  PERFORM UPDATE-STATUS.
```

where *recName* refers to the record type of CUR-*setName*.

- **FIND5RS**: the translation of the `FIND CURRENT` *recName* `WITHIN` *setName* is similar to the one of the FIND5S statement. The only difference is the following: if the specified *recName* does not correspond to the record type of the current record of *setName*, the exception status code 0503300 (*current record not of correct type*) is returned.

```
FIND5RS-recName-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0 OR 3
     MOVE 0503100 TO STATUS
     GO TO END-FIND5RS-recName-setName
  END-IF.
  IF recordType(CUR-setName) NOT = recName
     MOVE 0503300 TO STATUS
     GO TO END-FIND5RS-recName-setName
  END-IF.
```

```
    MOVE CUR-setName TO WK-recName.
    PERFORM MOVE-WK-recName-TO-CUR.
  END-FIND5RS-recName-setName.
    PERFORM UPDATE-STATUS.
```

where `recordType(CUR-`*setName*`)` refers to the record type of `CUR-`*setName*.

Format 6

Syntax `FIND OWNER WITHIN` *setName*

Effect The record identified is the owner of the set occurrence of the current record of *setName*.

SQL translation If the current record of *setName* is null (INIT-FLAG-*setName* = 0), the exception status code 0503100 is returned. If the current record of *setName* is a member record (`INIT-FLAG-`*setName* $\in \{1, 3, 4\}$), the owner record to be returned can be identified based on the value of the foreign key translating the set type. If the current record of the set already is the owner (`INIT-FLAG-`*setName* `= 3 OR 5`), the translation simply makes it the current record of the run unit and of its record type, similarly to the corresponding FIND5S statement.

```
FIND6-setName.
  MOVE ZERO TO STATUS.
  IF INIT-FLAG-setName = 0
     MOVE 0503100 TO STATUS
     GO TO END-FIND6-setName
  END-IF.
  IF INIT-FLAG-setName = 1 OR 3 OR 4
     EXEC SQL
        SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
        INTO :WK-recName-dbkey, :WK-recName-$f_1$, ... :WK-recName-$f_n$
        FROM $t_{recName}$
        WHERE $id_{setName}$ = :CUR-setName-$fk_{setName}$
     END-EXEC.
     IF SQLCODE = ZERO
        PERFORM MOVE-WK-recName-TO-CUR
     END-IF
  END-IF.
  IF INIT-FLAG-setName = 3 OR 5
     MOVE CUR-setName TO WK-recName
     PERFORM MOVE-WK-recName-TO-CUR
  END-IF.
END-FIND6-setName.
  PERFORM UPDATE-STATUS.
```

where *recName* is the owner record type of *setName*.

Format 7

Syntax `FIND` *recName* `WITHIN` *setName* `[ USING` $field_1$ `[,` $field_2$`] ...]`

Effect The record identified has a type equal to that of the record referenced by *recName*.

If the `USING` phrase is not specified, the record identified is the first member of the set *setName* according to the set ordering criteria for that set (like the FIND4FR statement).

If the `USING` phrase is specified, the record identified has the contents of the data items referenced by $field_1$, $field_2$,... equal to these data items in the UWA. So, in this case, the FIND7 statement is similar to the FIND3 statement, except that the reference values of $field_1$, $field_2$,... are not those of the current record of *setName*, but those of the UWA.

SQL translation The SQL translation of this statement is not provided below since it does not necessitate anything new compared to the FIND3 and FIND4FR statements.

Format 8

Syntax `FIND [{ FIRST |NEXT }]` *recName* `USING` *keyName*

Rules *keyName* must be defined in a key description entry in the subschema. *recName* must be defined in the subschema and must be a record type whose record key is identified by *keyName*.

Effect Records are maintained in ascending order of sequence by record key content for each *keyName* specified for record types in the subschema.

- **FIND8R**: `FIND` *recName* `USING` *keyName*
  The value of the data item in the record area for *recName*, which constitutes the record key referred to by *keyName*, is used to identify the record of type *recName* to be selected. If a record of the named record type with the specified key values is not found, two cases are possible.

  1. There are no records of the named record type whose key value is greater than the specified key value. In this case, the exception status code 0502400 (*no record found to satisfy record selection criteria*) is returned and currency indicators are left unchanged.
  2. A record of the named record type whose key value is greater than the specified key value is found. In this case, the execution of the statement *is successful* but the exception status code 0502401 (*record returned whose key value is greater than the specified key value*) is returned.

- **FIND8FR**: `FIND FIRST` *recName* `USING` *keyName*
  The record selected is the record whose record key value is the lowest within the logical ordering of records specified for *keyName*. If such a record is not found, the exception status code 0502100 is returned and the currency indicators are left unchanged.

- **FIND8NR**: `FIND NEXT` *recName* `USING` *keyName*
  The record selected is the record whose record key value is next higher relative to the current record of record key *keyName*. If such a record is not found, the exception status code 0502100 is returned and the currency indicators are left unchanged.

Upon successful completion of the FIND8 statement, the record identified is established as the current record of *keyName*, of the run unit, of its record type and of its set types. Currency indicators for all other record keys remain unchanged.

Assumption We assume here that *recName* is the only record type associated to key type *keyName*.

SQL translation The translation of the FIND8 statement makes use of two cursors, which are similar to the *order by* and *not less* cursors used in the COBOL-to-SQL translation rules of Chapter 7. Let $f_{1_{key}}...f_{p_{key}}$ be the fields of record type *recName* on top of which key type *keyName* is defined.

The `ORDER_BY_`*keyName* cursor selects all the records of type *recName* ordered in ascending order of the record key value *keyName*:

```
EXEC SQL
  DECLARE ORDER_BY_keyName CURSOR FOR
  SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
  FROM $t_{recName}$
  ORDER BY $c_{f_{1_{key}}}$, ... $c_{f_{p_{key}}}$
END-EXEC.
```

The `NOT_LESS_`*keyName* cursor selects all the records of type *recName* whose record key value *keyName* is higher or equal relative to the working record of *recName*. The records of this cursor are ordered in ascending order of the record key value.

```
EXEC SQL
  DECLARE NOT_LESS_keyName CURSOR FOR
  SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
  FROM $t_{recName}$
  WHERE $c_{f_{1_{key}}}$ > :WK-recName-$f_{1_{key}}$ OR
        ($c_{f_{1_{key}}}$ = :WK-recName-$f_{1_{key}}$ AND $c_{f_{2_{key}}}$ > :WK-recName-$f_{2_{key}}$) OR
        ...
        ($c_{f_{1_{key}}}$ = :WK-recName-$f_{1_{key}}$ AND
        ...
```

```
      $c_{f_{p-1_{key}}}$ = :WK-$recName$-$f_{p-1_{key}}$ AND $c_{f_{p_{key}}}$ >= :WK-$recName$-$f_{p_{key}}$)
  ORDER BY $c_{f_{1_{key}}}$,... $c_{f_{p_{key}}}$
END-EXEC.
```

- **FIND8R**: The SQL translation of the FIND8R statement basically consists in opening and fetching the NOT_LESS_*keyName* cursor, before doing a test on the record key value of the record obtained.

```
FIND8R-$keyName$.
  PERFORM CLOSE-LAST-CURSOR-$keyName$.
  MOVE UWA-$recName$-$f_{1_{key}}$ TO WK-$recName$-$f_{1_{key}}$.
  ...
  MOVE UWA-$recName$-$f_{p_{key}}$ TO WK-$recName$-$f_{p_{key}}$.
  EXEC SQL
     OPEN NOT_LESS_$keyName$
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-FIND8R-$keyName$
  END-IF.
  EXEC SQL
     FETCH NOT_LESS_$keyName$
     INTO :WK-$recName$-dbkey, :WK-$recName$-$f_1$, ... :WK-$recName$-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
     MOVE 0502400 TO STATUS
     PERFORM CLOSE-LAST-CURSOR-$recName$
     GO TO END-FIND8R-$keyName$
  END-IF.
  IF (WK-$recName$-$f_{1_{key}}$ = UWA-$recName$-$f_{1_{key}}$) AND
     ...
     (WK-$recName$-$f_{p_{key}}$ = UWA-$recName$-$f_{p_{key}}$)
        MOVE ZERO TO STATUS
  ELSE
        MOVE 0502401 TO STATUS
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
  PERFORM MOVE-WK-$recName$-TO-CUR-$keyName$.
END-FIND8R-$keyName$.
  PERFORM UPDATE-STATUS.
```

If the end of the cursor is reached, the exception status code 0502400 (*no record found to satisfy record selection criteria*) is returned. If a record with the proper record key value is found, the normal status code 0 is returned. If a record with a higher record key value is found, the exception status code 0502401 is returned.

- **FIND8FR**: The FIND8FR statement can be easily simulated by opening and fetching the ORDER_BY_*keyName* cursor. Indeed, the record whose record key value is the lowest is located at the first position of the cursor. If the end of the cursor is reached, the exception status code 0502100 is returned.

```
FIND8FR-keyName.
  MOVE ZERO TO STATUS.
  PERFORM CLOSE-LAST-CURSOR-keyName.
  EXEC SQL
     OPEN ORDER_BY_keyName
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-FIND8FR-keyName
  END-IF.
  EXEC SQL
     FETCH ORDER_BY_keyName
     INTO :WK-recName-dbkey, :WK-recName-$f_1$,...  :WK-recName-$f_n$
  END-EXEC.
  IF SQLCODE NOT = ZERO
     MOVE 0502100 TO STATUS
     PERFORM CLOSE-LAST-CURSOR-recName
     GO TO END-FIND8FR-keyName
  END-IF.
  PERFORM MOVE-WK-recName-TO-CUR.
  PERFORM MOVE-WK-recName-TO-CUR-keyName.
END-FIND8FR-keyName.
  PERFORM UPDATE-STATUS.
```

- **FIND8NR**: Translating the FIND8NR statement consists in fetching the last opened cursor for key type *keyName*, either `ORDER_BY_`*keyName* or `NOT_LESS_`*keyName*. If both cursors are closed, the exception 0503100 (*current record of key type is null*) is returned.

```
FIND8NR-keyName.
  MOVE ZERO TO STATUS.
  IF keyName-CURSOR-NONE
     MOVE 0503100 TO STATUS
     GO TO END-FIND8NR-keyName
  END-IF.
  IF keyName-CURSOR-NOT-LESS
     EXEC SQL
        FETCH NOT_LESS_keyName
        INTO :WK-recName-dbkey, :WK-recName-$f_1$,...  :WK-recName-$f_n$
     END-EXEC
  END-IF.
  IF keyName-CURSOR-ORDER-BY
     EXEC SQL
        FETCH ORDER_BY_keyName
        INTO :WK-recName-dbkey, :WK-recName-$f_1$,...  :WK-recName-$f_n$
     END-EXEC
```

```
      END-IF.
      IF SQLCODE NOT = ZERO
         MOVE 0502100 TO STATUS
         GO TO END-FIND8NR-keyName
      END-IF.
      PERFORM MOVE-WK-recName-TO-CUR.
      PERFORM MOVE-WK-recName-TO-CUR-keyName.
    END-FIND8NR-keyName.
      PERFORM UPDATE-STATUS.
```

8.4.3 GET statement

Syntax GET [{ *recName* |*field*$_1$ [,*field*$_2$] ... }]

Effect Execution of a GET statement causes the DBMS to place all or part of the current record of the run unit in its associated record area in the UWA.

The three following variants can be identified:

- **GET0**: GET
  The entire contents of the current record of the run unit are moved to the UWA.

- **GETR**: GET *recName*
  This form is equivalent to the GET0 variant: all fields in the current record of the run unit (*cur*) are moved to the UWA. But if the specified *recName* does not correspond to the record type of *cur*, the exception status code 0803300 is returned (*current record of the run unit is not correct record type*).

- **GETF**: GET *field*$_1$ [,*field*$_2$] ...
  Only those data items listed (*field*$_1$ [,*field*$_2$] ...) are moved to the record area for the current record in the UWA. All other fields for the record remains unchanged.

SQL translation The simulation of those three statements is straightforward.

For the GET0 statement, it basically consists in moving the current record of the run unit *cur* to the corresponding record of the UWA. If *cur* is not initialized, the exception status code 0803200 (*current record of the run unit is null*) is returned.

```
GET.
  MOVE ZERO TO STATUS.
  IF cur = null
     MOVE 0803200 TO STATUS
  ELSE
     MOVE cur TO UWA-recName
  END-IF.
```

where *recName* is the record type of the current record of the run unit (*cur*).

The translation of the GETR variant is almost the same as for the GET0 statement. An additional test has to be performed regarding the specified record type *recName*, which must correspond to the record type of the current record of the run unit *cur*. If this is not the case, the exception status code 0803300 (*current record of the run unit is not of correct record type*) is returned.

```
GETR-recName.
  MOVE ZERO TO STATUS.
  IF cur = null
     MOVE 0803200 TO STATUS
     GO TO END-GETR-recName
  END-IF.
  IF recordType(cur) NOT = recName
     MOVE 0803300 TO STATUS
     GO TO END-GETR-recName
  ELSE
     MOVE cur TO UWA-recName
  END-IF.
END-GETR-recName.
  PERFORM UPDATE-STATUS.
```

where `recordType(`*cur*`)` denotes the record type of the current of the run unit *cur*.

In the case of the GETF statement, only the specified fields are moved to the corresponding record in the UWA. This can simulated as follows:

```
GETF.
  MOVE ZERO TO STATUS.
  IF cur = null
     MOVE 0803200 TO STATUS
  ELSE
     MOVE cur-field1 TO UWA-recName-field1
     MOVE cur-field2 TO UWA-recName-field2
     ...
  END-IF.
```

where *recName* is the record type of the current record of the run unit (*cur*).

8.4.4 STORE statement

Syntax `STORE` *recName*
`[RETAINING CURRENCY FOR { REALM |SETS |RECORD |{`$setName_1$ `[,`$setName_2$`] ...}]`

Effect The `STORE` statement causes a record to be inserted in the database. It also establishes that record as the current record of the run unit. The record referenced by *recName*:

- becomes a member of each set type in which *recName* has been declared to be an *automatic* member.
- is established as the owner of an empty set for each set type in which *recName* is defined as an owner.

If the `RETAINING` phrase is not specified, the record referenced by *recName* becomes the current record of its area, the current record of its record type, the current record of all set types in which it has been declared to be a tenant (owner or member), and the current of all record key types that are defined for the record.

If the `RETAINING` phrase is specified with:

- `REALM`: the area currency indicators are left unchanged;
- `SETS`: no set type currency indicators are changed;
- `RECORD`: the record type currency indicators remains unchanged;
- $setName_1, setName_2, ...$: the set currency indicators for the named sets are not changed.

Assumption We assume that set ownership is identified *by application*, i.e., the set occurence of a set type s in which a new member record is inserted is the one of the current record of s.

SQL translation The SQL translation of the `STORE` statement obviously makes use of an `INSERT` statement. In addition, three issues are to be taken into account *before* the record can be inserted:

1. For each set type s in which the inserted record is declared an *automatic* member, one needs to retrieve the identifier of the current owner of s in order to use it as foreign key value.
2. One has to make sure that the inserted record does respect the uniqueness constraints (calc key, etc..).
3. If the `DB-KEY` has been translated as a SQL column, a unique value of this column has to be produced beforehand.

This leads to the following pseudo-code procedure:

```
STORE-$recName$.
  MOVE ZERO TO STATUS.
  PERFORM MOVE-UWA-TO-WK-$recName$.
  MOVE ownerID(CUR-$S_1$) TO WK-$recName$-$fk_{S_1}$.
     ...
  MOVE ownerID(CUR-$S_k$) TO WK-$recName$-$fk_{S_k}$.
  EXEC SQL
     SELECT COUNT(*)
     INTO :MYCOUNTER
     FROM $t_{recName}$
     WHERE $c_{ufg_1}$ = :WK-$recName$-$ufg_1$ OR
           ...
           $c_{ufg_w}$ = :WK-$recName$-$ufg_w$
  END-EXEC
  IF MYCOUNTER NOT = ZERO
     MOVE 1505100 TO STATUS
     GO TO END-STORE-$recName$
  END-IF.
  EXEC SQL
     SELECT MAX(dbkey)
     INTO :WK-$recName$-dbkey
     FROM $t_{recName}$
  END-EXEC
  ADD 1 TO WK-$recName$-dbkey.
  EXEC SQL
     INSERT INTO $t_{recName}$
           (dbkey, $fk_{S_1}$, ...  $fk_{S_k}$, $c_{f_1}$, ...  $c_{f_n}$)
     VALUES
           (:WK-$recName$-dbkey,:WK-$recName$-$fk_{S_1}$,...  :WK-$recName$-$fk_{S_k}$,
            :WK-$recName$-$f_1$,...  :WK-$recName$-$f_n$)
  END-EXEC.
  IF SQLCODE NOT = ZERO
     MOVE 1505100 TO STATUS
     GO TO END-STORE-$recName$
  END-IF.
  PERFORM CLOSE-LAST-CURSOR-$recName$.
  PERFORM MOVE-WK-$recName$-TO-CUR.
  PERFORM MOVE-WK-$recName$-TO-CUR-$K_1$.
  ...
  PERFORM MOVE-WK-$recName$-TO-CUR-$K_m$.
END-STORE-$recName$.
  PERFORM UPDATE-STATUS.
```

where:

- ufg_1 ... ufg_w denote the w groups of fields of record type $recName$ that are declared *unique*;

- $S_1, \ldots S_k$ are the k set types in which record type $recName$ has been declared to be an automatic member;

- `ownerID(CUR-`S_i`)` denotes the identifier value of the owner of the current set occurence of set type S_i;

- $K_1, \ldots K_m$ are the m key types that are defined for the record type $recName$.

8.4.5 `MODIFY` statement

Format 1

Syntax `MODIFY` [*recName*]
[{`INCLUDING` |`ONLY`} {`ALL` |$setName_1$ [,$setName_2$]...} `MEMBERSHIP`]

Rules *recName*, if specified, must be the record type name of the current record of the run unit. The current record of the run unit must be defined in the schema DDL as a member of the set types referenced by $setName_1$, $setName_2$ ... , if any.

Effect The execution of a `MODIFY` statement alters the contents of the data items in a record and/or changes the set membership of the record. The contents of the data items in the database are replaced with the contents of the corresponding items in the UWA. The object of the statement is the current record of the run unit.

If the `ONLY` phrase is specified no data items are to be updated. If the `ONLY` phrase is not specified, all data items in the current record of the run unit are to be updated.

If no `INCLUDING` or `ONLY` phrase is present, no set membership is changed. If an `INCLUDING` or `ONLY` phrase is specified with:

- `ALL`: the record's membership is changed in every set in which it is a member.
- $setName_1$, $setName_2$,...: the record's membership is changed in these set types.

In addition, the current record of the run unit becomes the current record of its area, the current record of its record type, and the current record of all sets in which it is a tenant.

Assumption As for the `STORE` statement, we assume that set ownership is identified *by application*, i.e., the set occurence of a set type *s* to which a member record is moved is the one of the current record of *s*.

SQL translation As for the `STORE` statement, some verification queries are needed in order to make sure that the new field values provided respect the uniqueness constraints. For the sake of clarity, we will ignore them in the provided translations.

The translation of format 1 of the MODIFY statement depends on the variant used:

- **MODIFYR**: `MODIFY` *recName*
  Without `ONLY` nor `INCLUDING` phrase, the translation of the statement is as follows. One first checks that the specified *recName* does correspond to the record type of the current record of the run unit (*cur*). If this is not the case,

the exception status code 1103300 is returned. If this is the case, all the data items of the record are then modified according to the values of the UWA fields. Then the modified record becomes the current record.

```
MODIFY-$recName$.
  MOVE ZERO TO STATUS.
  IF recordType($cur$) NOT = $recName$
     MOVE 1103300 TO STATUS
     GO TO END-MODIFY-$recName$
  END-IF.
  MOVE $cur$ TO WK-$recName$.
  PERFORM MOVE-UWA-TO-WK-$recName$.
  EXEC SQL
     UPDATE $t_{recName}$
     SET $c_{f_1}$ = :WK-$recName$-$f_1$,
         ...
         $c_{f_n}$ = :WK-$recName$-$f_n$,
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-MODIFY-$recName$
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
END-MODIFY-$recName$.
  PERFORM UPDATE-STATUS.
```

where `recordType(`*cur*`)` denotes the record type of the current of the run unit *cur*.

- **MODIFY0**: `MODIFY`
  This variant is similar to MODIFYR, where *recName* is implicitly the record type of the current record of the run unit.

```
MODIFY0.
  PERFORM MODIFYR-$recName$.
```

where *recName* denotes the record type of the current of the run unit *cur*.

- **MODIFYRIA**: `MODIFY` *recName* `INCLUDING ALL MEMBERSHIP`
  In the presence of an `INCLUDING ALL` phrase, the behaviour is similar except that the set membership of the member record are also changed. This translates in updating the values of the corresponding foreign keys such that they now refer to the owners of the current set occurences.

```
MODIFYRIA-$recName$.
  MOVE ZERO TO STATUS.
  IF recordType($cur$) NOT = $recName$
```

```
      MOVE 1103300 TO STATUS
      GO TO END-MODIFYRIA-$recName$
   END-IF.
   MOVE $cur$ TO WK-$recName$.
   PERFORM MOVE-UWA-TO-WK-$recName$.
   MOVE ownerID(CUR-$S_1$) TO WK-$recName$-$fk_{S_1}$.
      ...
   MOVE ownerID(CUR-$S_k$) TO WK-$recName$-$fk_{S_k}$.
   EXEC SQL
      UPDATE $t_{recName}$
      SET $fk_{S_1}$ = :WK-$recName$-$fk_{S_1}$,
          ...
          $fk_{S_k}$ = :WK-$recName$-$fk_{S_k}$,
          $c_{f_1}$ = :WK-$recName$-$f_1$,
          ...
          $c_{f_n}$ = :WK-$recName$-$f_n$,
      WHERE dbkey = :$cur$-dbkey
   END-EXEC.
   IF SQLCODE NOT = ZERO
      GO TO END-MODIFYRIA-$recName$
   END-IF.
   PERFORM MOVE-WK-$recName$-TO-CUR.
END-MODIFYRIA-$recName$.
   PERFORM UPDATE-STATUS.
```

where:

- `recordType(`*cur*`)` denotes the record type of the current of the run unit *cur*.
- $S_1, \ldots S_k$ are the k set types in which record type *recName* has been declared to be a member.
- `ownerID(CUR-`S_i`)` denotes the identifier value of the owner of the current set occurence of set type S_i;

- **MODIFYRIS**: `MODIFY` *recName* `INCLUDING` $setName_1$, $setName_2$ ... `MEMBERSHIP` The MODIFYRIS statement is simulated in the same way as MODIFYRIA, except that only the foreign keys translating set types $setName_1$, $setName_2$ ... are updated.

```
MODIFYRIS-$recName$.
   MOVE ZERO TO STATUS.
   IF recordType($cur$) NOT = $recName$
      MOVE 1103300 TO STATUS
      GO TO END-MODIFYRIS-$recName$
   END-IF.
   MOVE $cur$ TO WK-$recName$.
   PERFORM MOVE-UWA-TO-WK-$recName$.
   MOVE ownerID(CUR-$setName_1$) TO WK-$recName$-$fk_{setName_1}$.
```

```
  MOVE ownerID(CUR-$setName_2$) TO WK-$recName$-$fk_{setName_2}$.
     ...
  EXEC SQL
     UPDATE $t_{recName}$
     SET $fk_{setName_1}$ = :WK-$recName$-$fk_{setName_1}$,
         $fk_{setName_2}$ = :WK-$recName$-$fk_{setName_2}$,
         ...
         $c_{f_1}$ = :WK-$recName$-$f_1$,
         ...
         $c_{f_n}$ = :WK-$recName$-$f_n$,
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-MODIFYRIS-$recName$
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
END-MODIFYRIS-$recName$.
  PERFORM UPDATE-STATUS.
```

where:

- `recordType(`*cur*`)` denotes the record type of the current of the run unit *cur*.
- `ownerID(CUR-`S_i`)` denotes the identifier value of the owner of the current set occurence of set type S_i;

- **MODIFYROA**: `MODIFY` *recName* `ONLY ALL MEMBERSHIP`
The MODIFYROA variant is similar to MODIFYRIA, but only the set foreign keys are updated while the other columns are left unchanged.

```
MODIFYROA-$recName$.
  MOVE ZERO TO STATUS.
  IF recordType($cur$) NOT = $recName$
     MOVE 1103300 TO STATUS
     GO TO END-MODIFYROA-$recName$
  END-IF.
  MOVE $cur$ TO WK-$recName$.
  MOVE ownerID(CUR-$S_1$) TO WK-$recName$-$fk_{S_1}$.
     ...
  MOVE ownerID(CUR-$S_k$) TO WK-$recName$-$fk_{S_k}$.
  EXEC SQL
     UPDATE $t_{recName}$
     SET $fk_{S_1}$ = :WK-$recName$-$fk_{S_1}$,
         ...
         $fk_{S_k}$ = :WK-$recName$-$fk_{S_k}$
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
```

```
      GO TO END-MODIFY-recName
    END-IF.
    PERFORM MOVE-WK-recName-TO-CUR.
END-MODIFYROA-recName.
    PERFORM UPDATE-STATUS.
```

where:

- recordType(cur) denotes the record type of the current of the run unit cur.
- $S_1, \ldots S_k$ are the k set types in which record type $recName$ has been declared to be a member.
- ownerID(CUR-S_i) denotes the identifier value of the owner of the current set occurence of set type S_i;

- **MODIFYROS**: MODIFY *recName* ONLY $setName_1$, $setName_2$ ... MEMBERSHIP
  The MODIFYROS variant is similar to MODIFYROA, but only the set foreign keys corresponding to $setName_1$, $setName_2$ ... are updated while the other columns are left unchanged.

```
MODIFYROS-recName.
    MOVE ZERO TO STATUS.
    IF recordType(cur) NOT = recName
      MOVE 1103300 TO STATUS
      GO TO END-MODIFYROS-recName
    END-IF.
    MOVE cur TO WK-recName.
    MOVE ownerID(CUR-$setName_1$) TO WK-recName-$fk_{setName_1}$.
    MOVE ownerID(CUR-$setName_2$) TO WK-recName-$fk_{setName_2}$.
      ...
    EXEC SQL
      UPDATE $t_{recName}$
      SET $fk_{setName_1}$ = :WK-recName-$fk_{setName_1}$,
          $fk_{setName_2}$ = :WK-recName-$fk_{setName_2}$
          ...
      WHERE dbkey = :cur-dbkey
    END-EXEC.
    IF SQLCODE NOT = ZERO
      GO TO END-MODIFY-recName
    END-IF.
    PERFORM MOVE-WK-recName-TO-CUR.
END-MODIFYROS-recName.
    PERFORM UPDATE-STATUS.
```

where:

- recordType(cur) denotes the record type of the current of the run unit cur.

– `ownerID(CUR-`$setName_i$`)` denotes the identifier value of the owner of the current set occurence of set type $setName_i$;

Format 2

Syntax `MODIFY` $field_1$ `[,`$field_2$`]...`
`[INCLUDING {ALL |`$setName_1$ `[,`$setName_2$`]...} MEMBERSHIP]`

Rules $field_1$, $field_2$,... must reference data items contained within the current record of the run unit. The current record of the run unit must be defined in the schema DDL as a member of the set types referenced by $setName_1$, $setName_2$ ... , if any.

Assumption As for the `STORE` statement, we assume that set ownership is identified *by application*, i.e., the set occurence of a set type s to which a member record is moved is the one of the current record of s.

Effect The `MODIFY` statement alters the contents of $field_1$, $field_2$, ... in a record and/or changes the set membership of the record. The object of the statement is the current record of the run unit.

If no `INCLUDING` phrase is present, no set membership is changed. If an `INCLUDING` phrase is specified with:

- `ALL`: the record's membership is changed in every set in which it is a member.
- $setName_1$, $setName_2$,...: the record's membership is changed in these set types.

In addition, the current record of the run unit becomes the current record of its area, the current record of its record type, and the current record of all sets in which it is a tenant.

SQL translation As for the `STORE` statement, some verification queries are needed in order to make sure that the new field values provided respect the uniqueness constraints. For the sake of clarity, we will ignore them in the provided translations.

The translation of format 2 of the MODIFY statement depends on the variant used:

- **MODIFYF**: `MODIFY` $field_1$, $field_2$ ...
  This variant causes the data items $field_1$, $field_2$, ... of the current record of the run unit *cus* to be updated.

```
MODIFYF.
  MOVE ZERO TO STATUS.
  MOVE cur TO WK-recName.
  MOVE UWA-$recName$-$field_1$ TO WK-$recName$-$field_1$.
  MOVE UWA-$recName$-$field_2$ TO WK-$recName$-$field_2$.
     ...
  EXEC SQL
     UPDATE $t_{recName}$
     SET $c_{field_1}$ = :WK-$recName$-$field_1$,
         $c_{field_2}$ = :WK-$recName$-$field_2$,
         ...
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-MODIFYF
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
END-MODIFYF
  PERFORM UPDATE-STATUS.
```

where $recName$ denotes the record type of the current record of the run unit cur.

- **MODIFYFIA**: `MODIFY` $field_1$, $field_2$ ... `INCLUDING ALL MEMBERSHIP` The MODIFYFIA variant is similar to the MODIFYF form where, in addition, the memberships of the current record are updated for each set in which the record is a member. This translates as follows:

```
MODIFYFIA.
  MOVE ZERO TO STATUS.
  MOVE $cur$ TO WK-$recName$.
  MOVE UWA-$recName$-$field_1$ TO WK-$recName$-$field_1$.
  MOVE UWA-$recName$-$field_2$ TO WK-$recName$-$field_2$.
     ...
  MOVE ownerID(CUR-$S_1$) TO WK-$recName$-$fk_{S_1}$.
     ...
  MOVE ownerID(CUR-$S_k$) TO WK-$recName$-$fk_{S_k}$.
  EXEC SQL
     UPDATE $t_{recName}$
     SET $c_{field_1}$ = :WK-$recName$-$field_1$,
         $c_{field_2}$ = :WK-$recName$-$field_2$,
         ...
         $fk_{S_1}$ = :WK-$recName$-$fk_{S_1}$,
         ...
         $fk_{S_k}$ = :WK-$recName$-$fk_{S_k}$
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-MODIFYFIA
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
```

```
END-MODIFYFIA
  PERFORM UPDATE-STATUS.
```

where:

- $recName$ denotes the record type of the current record of the run unit cur.
- $S_1, \ldots S_k$ are the k set types in which record type $recName$ has been declared to be a member.
- `ownerID(CUR-`S_i`)` denotes the identifier value of the owner of the current set occurence of set type S_i;

- **MODIFYFIS**: `MODIFY` $field_1$, $field_2$ ... `INCLUDING` $setName_1$, $setName_2$, ... `MEMBERSHIP`
This variant is equivalent to the MODIFYFIA statement, except that the record set memberships are changed only for set types $setName_1$, $setName_2$, ... The other set memberships remain unchanged.

```
MODIFYFIS.
  MOVE ZERO TO STATUS.
  MOVE $cur$ TO WK-$recName$.
  MOVE UWA-$recName$-$field_1$ TO WK-$recName$-$field_1$.
  MOVE UWA-$recName$-$field_2$ TO WK-$recName$-$field_2$.
     ...
  MOVE ownerID(CUR-$setName_1$) TO WK-$recName$-$fk_{setName_1}$.
  MOVE ownerID(CUR-$setName_2$) TO WK-$recName$-$fk_{setName_2}$.
     ...
  EXEC SQL
     UPDATE $t_{recName}$
     SET $c_{field_1}$ = :WK-$recName$-$field_1$,
         $c_{field_2}$ = :WK-$recName$-$field_2$,
         ...
         $fk_{setName_1}$ = :WK-$recName$-$fk_{setName_1}$,
         ...
         $fk_{setName_k}$ = :WK-$recName$-$fk_{setName_k}$
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE NOT = ZERO
     GO TO END-MODIFYFIS
  END-IF.
  PERFORM MOVE-WK-$recName$-TO-CUR.
END-MODIFYFIS
  PERFORM UPDATE-STATUS.
```

where:

- $recName$ denotes the record type of the current record of the run unit cur.

  - `ownerID(CUR-`S_i`)` denotes the identifier value of the owner of the current set occurence of set type S_i;

8.4.6 `ERASE` statement

Syntax `ERASE [`*recName*`] [ALL MEMBERS]`

Rules If specified, *recName* must be the record type of the current record of the run unit.

Effect Execution of the `ERASE` statement causes one or more records to be removed from the database.

If the `ALL` phrase is not specified, two cases are considered:

- the current record of the run unit is not the owner of a set that currently has members. In this case, the record is removed from the database and disconnected from all sets of which it is a member.
- the current record of the run unit is the owner of a set that currently has members. Then, an exception condition results.

If the `ALL` phrase is specified and the current record of the run unit is the owner of a set that currently has members all of these member records are also removed from the database. Any record so removed is treated as it were the record subject of the `ERASE` statement. The process is repeated until all records have been removed from the database that are hierarchically related to the current record of the run unit. If the current record of the run unit is not the owner of a set that currently has members, execution proceeds as if the `ALL` phrase had not been specified. If any erased record is currently the member of another set, the record is obviously disconnected from that set.

Currency indicators are affected as follows:

- The current record of the run unit is nulled.
- If any record removed from the database is the current record of its record type, the currency indicator for the record type is nulled.
- If any record removed from the database is the current record of a set type that it is the owner of, the currency indicator for the set type is nulled.
- If any record removed from the database is the current record of a set type that it is a member of, the currency indicator for the set type is updated to identify the position between the two records where the removed record had been.
- If an entire set occurrence is removed from the data base as a result of an `ERASE ALL` statement and the current record of the set type was one of the records removed, the currency indicator for the set type is nulled.

- If any record removed from the data base is the current record of its area, the currency indicator for the area is updated to identify the position between the two database keys where the removed record had been.
- If any record from the database is the current record of a record key type, the currency indicator for the key type is updated to identify the position between the two records where the removed record had been.

Note that the current record of a set type, even when deleted, remains valid for subsequent DML statements requiring a current record of a set (except `FIND CURRENT WITHIN` $setName$). For example, the following sequence is valid:

```
ERASE B.
FIND NEXT WITHIN A-B.
```

where `B` is the member record type of set type `A-B`.

SQL translation

- **ERASE0**: `ERASE`
  This variant is translated as follows. If the current record of the run unit *cur* is null, the exception code 0403200 (*current record of the run unit is null*) is returned. If *cur* is the owner of a set that currently has members, the exception status code 0407200 (*deletion of a non-empty set was requested* is returned. If *cur* is the current record of a set type $setO_i$ that it is an owner of, the current record of the set is nulled and the corresponding set currency flag `INIT-FLAG-`$setO_i$ is set to 3 (*current record of the set was deleted*). If *cur* is the current record of a set type $setM_i$ that it is a member of, its member position in its set occurence is recorded and the corresponding set currency flag `INIT-FLAG-`$setO_i$ is set to 3 (*current record of the set was deleted*)

```
ERASE.
  MOVE ZERO TO STATUS.
  IF $cur$ = null
     MOVE 0403200 TO STATUS
     GO TO END-ERASE
  END-IF.
  EXEC SQL
     SELECT COUNT(*)
     INTO :NBR-MEMBERS
     FROM $t_{memRec(setO_1)}$
     WHERE $fk_{setO_1}$ = :$cur$-$id_{setO_1}$
  END-EXEC.
  IF NBR-MEMBERS > 0
     MOVE 0407200 TO STATUS
     GO TO END-ERASE
  END-IF.
  ...
  EXEC SQL
```

```
     SELECT COUNT(*)
     INTO :NBR-MEMBERS
     FROM $t_{memRec(setO_k)}$
     WHERE $fk_{setO_k}$ = :$cur$-$id_{setO_k}$
  END-EXEC.
  IF NBR-MEMBERS > 0
     MOVE 0407200 TO STATUS
     GO TO END-ERASE
  END-IF.
  EXEC SQL
     DELETE FROM $t_{recName}$
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE = ZERO
     IF CUR-$setO_1$ = $cur$
        MOVE null TO CUR-$setO_1$
        MOVE 3 TO INIT-FLAG-$setO_1$
     END-IF
     ...
     IF CUR-$setO_k$ = $cur$
        MOVE null TO CUR-$setO_k$
        MOVE 3 TO INIT-FLAG-$setO_k$
     END-IF
     IF CUR-$setM_1$ = $cur$
        PERFORM RECORD-POSITION-CUR-$setM_1$
        MOVE 3 TO INIT-FLAG-$setM_1$
     END-IF
     ...
     IF CUR-$setM_w$ = $cur$
        PERFORM RECORD-POSITION-CUR-$setM_w$
        MOVE 3 TO INIT-FLAG-$setM_w$
     END-IF
  ELSE
     MOVE 0407200 TO STATUS
  END-IF
END-ERASE.
  PERFORM UPDATE-STATUS.
```

where:

- $recName$ denotes the record type of the current record of the run unit cur.
- $memRec(setName)$ denotes the member record type of set type $setName$.
- $setO_1$ ... $setO_k$ are the k set types in which the current record of the run unit is an owner.
- $setM_1$ ... $setM_w$ are the w set types in which the current record of the run unit is an member.

- procedure RECORD-POSITION-CUR-*setName* allows to record the position of the current (member) record of set type *setName* in its set occurence. This can be using the following query:

```
EXEC SQL
  SELECT COUNT(*)
  INTO :POSITION-CUR-$setName$
  FROM $t_{recName}$
  WHERE  $orderCriteria_{setName}$ <= CUR-$setName$-$orderCriteria_{setName}$ AND
         $fk_{setName}$ = :CUR-$setName$-$fk_{setName}$
END-EXEC
```

- **ERASER**: ERASE *recName*
  The SQL translation of this variant is very similar to the one of the **ERASE0** statement. An additional test checks that *recName* corresponds to the record type of the current record of the run unit. If this is not the case, the exception status code 0403300 (*current record of the run unit is not of correct record type*) is returned.

- **ERASEALL**: ERASE ALL MEMBERS
  The SQL translation of the ERASEALL variant is much more complicated. It makes use of cursors of the following form, which selects, for a particular set type *setName*, the member records attached to a given owner to be deleted:

```
EXEC SQL
   DECLARE $setName$_MEMBERS_TO_ERASE CURSOR FOR
   SELECT dbkey, $c_{f_1}$, ... $c_{f_n}$
   FROM $t_{memRec(setName)}$
   WHERE $fk_{setName}$ = :WK-$ownerRecord(setName)$-$id_{setName}$
END-EXEC.
```

  where:

  - *memRec*(*setName*) denotes the member record type associated to set type *setName*.
  - *ownerRecord*(*setName*) denotes the owner record type associated to set type *setName*.

In addition, a recursive procedure is used for removing all records from the database that are hierarchically related to the a given owner of a set type *setName*.

```
ERASE-MEMBERS-OF($setName$, ownerID).
   MOVE ownerID to = :WK-$ownerRecord(setName)$-$id_{setName}$.
   EXEC SQL
      OPEN $setName$_MEMBERS_TO_ERASE
   END-EXEC.
   PERFORM UNTIL SQLCODE NOT = ZERO
      EXEC SQL
      FETCH $setName$_MEMBERS_TO_ERASE
      INTO :WK-$memRec(setName)$-dbkey,
         :WK-$memRec(setName)$-$f_1$,
          ...
         :WK-$memRec(setName)$-$f_n$
   END-EXEC
   IF SQLCODE = ZERO
      ERASE-MEMBERS-OF($setO_1(memRec(setName))$,WK-$memRec(setName)$-$id_{setO_1}$)
      ...
      ERASE-MEMBERS-OF($setO_k(memRec(setName))$,WK-$memRec(setName)$-$id_{setO_k}$)
      EXEC SQL
         DELETE FROM $t_{memRec(setName)}$
         WHERE dbkey = :WK-$memRec(setName)$-dbkey
      END-EXEC
   END-IF
   IF SQLCODE = ZERO
      IF CUR-$setO_1(memRec(setName))$ = WK-$memRec(setName)$
         MOVE null TO CUR-$setO_1(memRec(setName))$
         MOVE 3 TO INIT-FLAG-$setO_1(memRec(setName))$
      END-IF
      ...
      IF CUR-$setO_k(memRec(setName))$ = WK-$memRec(setName)$
         MOVE null TO CUR-$setO_k(memRec(setName))$
         MOVE 3 TO INIT-FLAG-$setO_k(memRec(setName))$
      END-IF
      IF CUR-$setM_1(memRec(setName))$ = WK-$memRec(setName)$
         PERFORM RECORD-POSITION-CUR-$setM_1(memRec(setName))$
         MOVE 3 TO INIT-FLAG-$setM_1(memRec(setName))$
      END-IF
      ...
      IF CUR-$setM_w(memRec(setName))$ = WK-$memRec(setName)$
         PERFORM RECORD-POSITION-CUR-$setM_w(memRec(setName))$
         MOVE 3 TO INIT-FLAG-$setM_w(memRec(setName))$
      END-IF
   END-IF
END-PERFORM.
EXEC SQL
   CLOSE $setName$_MEMBERS_TO_ERASE
END-EXEC.
```

where:

- $memRec(setName)$ denotes the member record type associated to set type $setName$.

- $ownerRecord(setName)$ denotes the owner record type associated to set type $setName$.

- $setO_1(recName)$ ... $setO_k(recName)$ denote the k set types of which $recName$ is declared as the owner record type.
- $setM_1(recName)$ ... $setM_w(recName)$ denote the w set types of which $recName$ is declared as the member record type.

Consequently, the `ERASE ALL MEMBERS` statement can be simulated as follows:

```
ERASE-ALL-MEMBERS.
  MOVE ZERO TO STATUS.
  IF $cur$ = null
     MOVE 0403200 TO STATUS
     GO TO END-ERASE-ALL-MEMBERS
  END-IF.
  PERFORM ERASE-MEMBERS-OF($setO_1$, $cur$-$id_{setO_1}$).
  ...
  PERFORM ERASE-MEMBERS-OF($setO_p$, $cur$-$id_{setO_p}$).
  EXEC SQL
     DELETE FROM $t_{recName}$
     WHERE dbkey = :$cur$-dbkey
  END-EXEC.
  IF SQLCODE = ZERO
     IF CUR-$setO_1$ = $cur$
         MOVE null TO CUR-$setO_1$
         MOVE 3 TO INIT-FLAG-$setO_1$
     END-IF
     ...
     IF CUR-$setO_p$ = $cur$
         MOVE null TO CUR-$setO_p$
         MOVE 3 TO INIT-FLAG-$setO_p$
     END-IF
     IF CUR-$setM_1$ = $cur$
         PERFORM RECORD-POSITION-CUR-$setM_1$
         MOVE 3 TO INIT-FLAG-$setM_1$
     END-IF
     ...
     IF CUR-$setM_w$ = $cur$
         PERFORM RECORD-POSITION-CUR-$setM_w$
         MOVE 3 TO INIT-FLAG-$setM_w$
     END-IF
  ELSE
     MOVE 0407200 TO STATUS
  END-IF
END-ERASE-ALL-MEMBERS.
  PERFORM UPDATE-STATUS.
```

where:

- $recName$ denotes the record type of the current record of the run unit cur.

- $setO_1$ ... $setO_p$ are the p set types in which the current record of the run unit is an owner.
- $setM_1$ ... $setM_w$ are the w set types in which the current record of the run unit is an member.
- procedure `RECORD-POSITION-CUR-`$setName$ allows to record the position of the current (member) record of set type $setName$ in its set occurence.

8.5 Tool Support

This section describes the tools that support the migration approach developed in this chapter. The tool architecture is based on the combination of two complementary transformational environments, namely DB-MAIN (DB-MAIN, 2006) and the ASF+SDF Meta-Environment (van den Brand et al., 2001). Below, we describe a set of tools that support the different processes. Figure 8.6 depicts the tools that allow the automatic adaptation of the legacy COBOL programs. Those tools constitute our personal contribution.

Reverse engineering The inventory step is supported by four tools. The first one cleans the COBOL source code (e.g., removes the line numbers), resolves the copybooks and produces a report summarizing the missing programs and copybooks. The second tool parses the source code to check that all the code fragments can be analyzed and to list all program calls and data manipulation instructions. The third tool analyzes the JCL to find out which (physical) files are used by each program. Lastly, DB-MAIN allows to store and manipulate the call and usage graphs that are created from the report produced by the second tool.

The database reverse engineering process is also supported by a mixture of different tools. DB-MAIN is used to parse the DDL code, and to successively refine and conceptualize the database schema. Data analysis programs are automatically generated from the mapping that holds between the physical and logical database schemas. Those programs browse the legacy database and verify several properties by answering, among others, the following questions: (1) do the various data fields match their supposed type; (2) are filler items used as reserved space or do they hide relevant information;(3) do the data instances verify some explicit or potential constraints.

Schema conversion DB-MAIN offers general functions and components supporting the schema conversion phase, among which, (1) a generic model of schema representation based on the GER (Generic Entity/Relationship) model to describe data structures in all abstraction levels and according to all popular modelling paradigms; (2) a graphical interface to view the repository and apply operations; (3) a transformational toolbox rich enough to encompass most database engineering and reverse engineering processes; (4) A 4GL (*Voyager2*) as well as a Java API that allow analysts to quickly develop their own customized processors.

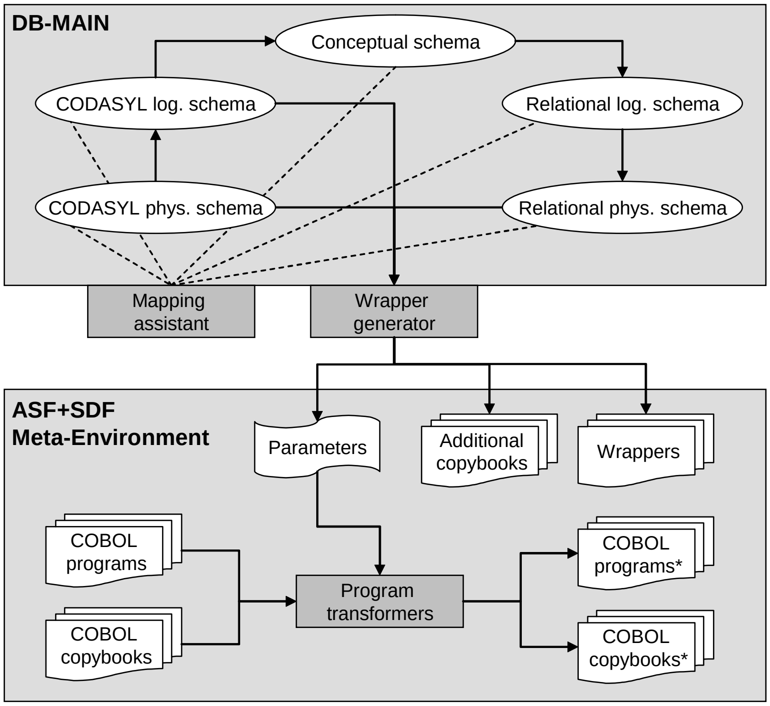


Figure 8.6: Tool architecture for program adaptation in CODASYL to relational migration.

Data migration The data migration process derives from the mapping that is maintained between the successive database schemas. We use a data migration program generator that takes such a mapping as input, and produces the corresponding data migration program. The generated data migrator reads the legacy database entirely, converts the data when necessary, and fills the new relational database.

Wrapper generation The wrapper generator is a plug-in of DB-MAIN written in Java. It takes as inputs (1) the legacy CODASYL (IDS/II) physical schema, (2) the refined CODASYL (IDS/II) logical schema, (3) the target relational schema, and (4) the mapping between these three schemas. The generation produces (1) a set of wrappers, each corresponding to a migrated record type, (2) a set of related copybooks (variable declarations and new code fragments), (3) the necessary input arguments of the program transformation tools. The generated wrappers are COBOL programs including embedded SQL commands. The wrapper generator allows the user to select the target database and execution platforms among the following:

- DBSP-DB2/COBOL Bull GCOS 8
- Oracle/COBOL Bull GCOS 7
- DB2/COBOL IBM
- PostgreSQL/COBOL Microfocus

Figure 8.7 shows a simplified code fragment allowing to generate a procedure that simulates a FIND4NR statement (`FIND NEXT` *recName* `WITHIN` *setName*).

Program transformation The conversion of the legacy application programs relies on the ASF+SDF Meta-Environment (van den Brand et al., 2001). We reused an SDF version of the IBM VS COBOL II grammar obtained by Lämmel and Verhoef (2001). We added an SDF module specifying the syntax of the CODASYL (IDS/II) statements. We then specified a set of rewrite rules (ASF equations) on top of this enriched grammar in order to obtain several program transformation tools. The first tool allows to convert IDS/II primitives into wrapper invocations[6]. Figure 8.8 provides an example ASF equation, which specifies the general conversion rule applied in Figure 8.5. The second tool supports the *refactoring* of the legacy programs, which mainly consists in the replacement of each IDS/II primitive with a procedure call[7], as the rewrite rule of Figure 8.9 illustrates. The invoked procedure executes/simulates the legacy primitive. The third tool is designed to convert programs that indirectly access the legacy database (via calls to other programs). The transformation tool chain also comprises lexical processors (typically Perl scripts) for source-code preprocessing, post-processing and pretty-printing.

[6] In the style of the *Wrapper strategy* (P1)

[7] In the style of the *Statement rewriting strategy* (P2)

```
public void genFind4NR(CobWriter wrapper, DBMEntityType idsRec,
     DBMRelationshipType set){
   String setName = set.getName();
   String recName = idsRec.getName();
   wrapper.printSectionHead("Find Next Rec Within Set <"+ setName + ">", "
       FIND4NR-" + setName);
   wrapper.printL("     MOVE ZERO TO WR-STATUS.");
   String setCursor = "FK" + setName;
   wrapper.printL("     IF INIT-FLAG-" + setName + " = 0");
   wrapper.printL("        MOVE 0503100 TO WR-STATUS");
   wrapper.printL("        GO TO END-FIND4NR-" + setName);
   wrapper.printL("     END-IF.");
   wrapper.printL("     IF INIT-FLAG-" + setName + " = 2 OR 5");
   wrapper.printL("         PERFORM FIND4FR-"+setName);
   wrapper.printL("         GO TO END-FIND4NR-" + setName);
   wrapper.printL("     END-IF.");
   wrapper.printL("     IF NOT(" + "LC-"+ setCursor + ") OR INIT-FLAG-"+ setName
        +" = 4");
   wrapper.printL("        PERFORM OPEN-AND-POSITION-"+setCursor);
   wrapper.printL("        IF SQLCODE NOT = ZERO");
   wrapper.printL("           MOVE 0502100 TO WR-STATUS");
   wrapper.printL("           GO TO END-FIND4NR-" + setName);
   wrapper.printL("        END-IF");
   wrapper.printL("     END-IF.");
   wrapper.printL("     PERFORM FETCH-" + setCursor + ".");
   wrapper.printL("     IF SQLCODE NOT = ZERO");
   wrapper.printL("        MOVE 0502100 TO WR-STATUS");
   wrapper.printL("        PERFORM CLOSE-LAST-CURSOR");
   wrapper.printL("        GO TO END-FIND4NR-" + setName);
   wrapper.printL("     END-IF.")
   wrapper.printL("     PERFORM MOVE-WK-TO-CUR.");
   wrapper.printL(" END-FIND4NR-" + setName + ".");
   wrapper.printL("     PERFORM UPDATE-WR-STATUS.");
}
```

Figure 8.7: A (simplified) code fragment of the wrapper generator.

```
[equ-FIND4N]
db-2-wrap(FIND NEXT WITHIN setName, record-types, set-member-tbl, key-names,
     area-list)
=  to-comment(FIND NEXT WITHIN setName)
   SET WR-ACTION-FIND4NR TO TRUE
   MOVE to-literal(memberRecordName) TO WR-CALL-NAME
   MOVE to-literal(setName) TO WR-OPTION
   PERFORM CALL-WRAPPER
when element(set-member-tbl, setName) == true,
     memberRecordName := lookup(set-member-tbl, setName)
```

Figure 8.8: A (simplified) ASF equation that rewrites a `FIND NEXT WITHIN` *setName* statement as a corresponding wrapper invocation.

```
[equ-FIND4N-bis]
db-2-proc(FIND NEXT WITHIN setName, record-types, set-member-tbl, key-names,
    area-list)
=  to-comment(FIND NEXT WITHIN setName)
   PERFORM to-paragraph(WR-FNXW, setName)
when element(set-member-tbl, setName) == true
```

Figure 8.9: A (simplified) ASF equation that rewrites a `FIND NEXT WITHIN` *setName* statement as a corresponding procedure call.

8.6 Related Work

The use of wrapping techniques is not new in software maintenance, particularly in the context of system migration. The use of wrappers have been proposed by many authors to support the migration of software systems towards various new architectures and platforms. The existing wrapping techniques concern different components of the legacy system including the graphical user interface, the underlying database or the legacy applications. The encapsulated components may belong to different levels of granularity (Sneed, 2000). Among the specific works on wrapping, Lucia et al. (2006) propose a practical approach to migrating legacy systems to multi-tier, web-based architectures. This approach consists in (1) migrating the graphical user interface and (2) restructuring and wrapping the original legacy code. In the challenging context of migration to SOA, Sneed (2006) presents a wrapping-based approach according to which legacy program functions are offered as web services to external users. As seen above, the major specificity of the wrappers discussed in this chapter is that they encapsulate a brand new system component (i.e., the target database) in order to reuse the legacy applications.

Concerning database reengineering, several approaches have been proposed in the literature. The approach by Jeusfeld and Johnen (1994) is divided into three parts: mapping of the original schema into a meta model, rearrangement of the intermediate representation and production of the target schema. The Varlet project (Jahnke and Wadsack, 1999) adopts a typical two phase reengineering process comprising a reverse engineering process phase followed by a standard database implementation. Bianchi et al. (2000) propose an iterative approach to database reengineering which eliminates the *ageing symptoms* of the legacy database (Visaggio, 2001) when incrementally migrating the latter towards a modern platform. Although all those approaches are, in some point, similar to the schema reengineering method discussed in this chapter, they mainly focus on the database conversion phase. Technical aspects of program conversion in the case of inter-paradigmatic database migration receive much more attention in the present chapter.

Some other research results address particular database migration scenarios, as we do in this chapter. Among those works, Menhoudj and Ou-Halima (1996) present a method to migrate the data of COBOL legacy system into a relational database management system. The hierarchical to relational database migration is discussed by Meier et al. (1994). General approaches to migrate relational database to OO technology are proposed by Behm et al. (1997) and Missaoui et al. (1998).

Finally, our approach to program transformation shares similar points with the work by Veerman 2004; 2006, in which maintenance transformations are applied to large legacy systems. In (Veerman, 2006), the author presents a method allowing to upgrade COBOL applications to a new version of the underlying database system. The transformation processs comprises several steps including pre-processing, transformation, post-processing and pretty-printing. The main rewrite rules are specified by means of the ASF+SDF Meta-Environment.

8.7 Conclusions

This chapter presented a tool-supported approach to migrating CODASYL databases towards a relational database platform. The discussion focused on the program conversion phase, by suggesting the use of backward data wrappers. We further adressed technical issues related to the proposal, and we specified a set of systematic translation rules for the simulation of CODASYL data manipulation statements in SQL.

Roadmap

Chapter 9 presents and discusses two industrial migration projects, for which the migration approach and tools described in the current chapter have been used.

Chapter 9

Industrial Migration Projects

Experience is a hard teacher because she
gives the test first, the lesson afterwards.
– Vernon Sanders Law

This chapter[1] presents the application of our migration approach and tools presented in Chapter 8 in the context of real-size industrial migration projects. Those projects were carried out in collaboration with ReVeR, a spin-off company originating from the database engineering laboratory of the University of Namur, that provides its customers with, among others, database reverse-engineering and reengineering services. The customer was the IT department of a Belgian federal ministry. For both projects, the author was personnally in charge of the program conversion phase.

9.1 Project 1: IDS/II to DB2

The goal of the first *proof-of-concept* project was to migrate a large COBOL system towards a relational (DB2) database platform. The legacy system runs on a Bull GCOS8 mainframe and is made of nearly 2 300 programs, totaling more than 2 million lines of COBOL code. The legacy applications make use of an IDS/II database. The source physical database schema comprises 231 record types, 213 sets and 648 fields. The target system is made up of a subset of the legacy application programs, which now accesses a DB2 database running through a DBSP gateway. DBSP is a database management service designed to enable GCOS8 applications to access relational databases on a remote system (BULL, 2001).

Below, we describe the process followed as well as the results obtained during the successive steps of the project.

[1] This chapter extends two industrial track papers. The first paper (Henrard et al., 2007) appeared in the proceedings of the 23rd International Conference on Software Maintenance (ICSM'07). The second paper (Henrard et al., 2008) was published in the proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08). Both papers are co-authored by Jean Henrard, Didier Roland and Jean-Luc Hainaut.

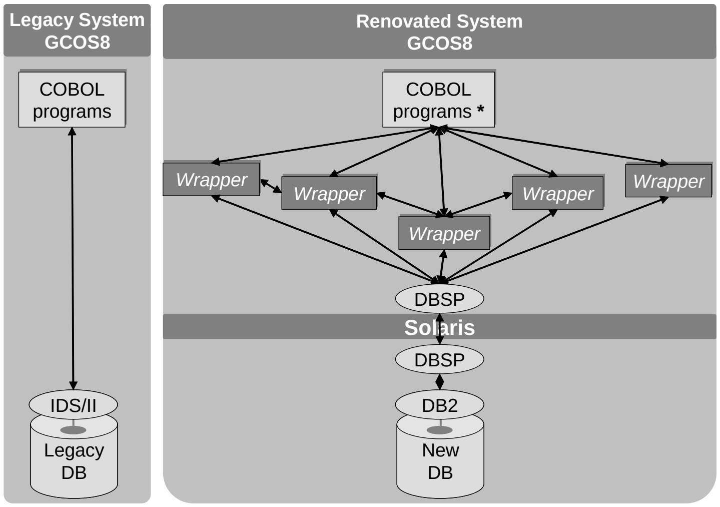


Figure 9.1: Project 1: general architecture.

Reverse engineering

The first phase was the reverse engineering of the system. This process was first performed on a small, consistent subset of the system in order to better understand the peculiarities of the system and to allow the customer to evaluate the results early in the process. Then the whole database has been reverse engineered gradually.

The inventory step allowed us to colllect all the necessary sources and to compute both the call and usage graphs. Although only a subset of the applications were selected for migration, the reverse engineering process analyzed the complete system, in order to obtain as precise results as possible. This involved a total of 2 273 programs, 64 809 program calls, 105 097 IDS/II verbs and 25 163 file access verbs. The resulting call graph contains 2 273 nodes (programs) and 9 527 call relationships. The usage graph contains 313 (programs) + 218 (records) nodes and 2 887 usage vertices.

During the database reverse engineering process, system dependency graphs analysis techniques, as those presented in Chapter 5, allowed us to identify the program variables used to manipulate database records, in order to deduce a more precise record decomposition. Dataflow analysis was also used to elicit implicit data dependencies that hold between database fields, among which potential foreign keys. The data validation step revealed that many implicit referential constraints were actually violated by the legacy data. This is explained by the fact that most of those constraints are simply encoding rules which are not always checked again when data are updated, and by the fact that users find tricks to bypass some rules.

	Physical schema (IDS/II)	Logical schema (IDS/II)	Conceptual schema	Relational schema (DB2)
# entity types	159	159	156	171
# rel. types	148	148	90	0
# attributes	458	9 027	2 176	2 118
max. # att/ent.type	8	104	61	94

Table 9.1: Project 1: Comparison of successive versions of the database schema.

	Transformed	Manually adapted	Source code size (kLOC)
# programs	669	17	705
# copybooks	3 917	68	35
# IDS/II inst.	5 314	420	-

Table 9.2: Project 1: Legacy program transformation results.

Schema conversion

Table 9.1 gives a comparison of the successive versions of the database schema. The physical IDS/II schema is the initial schema extracted from the DDL code (here we consider the subset actually migrated). The logical IDS/II schema is the physical schema with a finer-grained structure. It was obtained by resolving numerous copybooks in which structural decompositions of physical attributes are declared. In the logical schema, most attributes were declared several times through *redefines* clauses, hence the huge total number of attributes. The conceptual schema comprised only one declaration per attribute. When a conflict occured, the chosen attribute decomposition was the one considered as the most expressive by the analyst. In addition, the number of entity types is different since some technical record types were discarded while other ones were split (sub-types). Finally, the relational schema shows (1) an increase in the number of entity types due to the decomposition of arrays, and (2) a reduction of the number of attributes due to the aggregation of compound fields.

Wrapper-based program conversion

The wrapper generation phase produced 159 database wrappers, totalizing 450 thousands lines of code. The results obtained during the legacy code adaptation are summarized in Table 9.2. A total of 669 programs and 3 917 copybooks were actually converted. We notice that around 92% of the IDS/II verbs were transformed automatically, while the manual work concerned only 85 distinct source code files. The automated transformation of the legacy code took a bit less than 95 minutes.

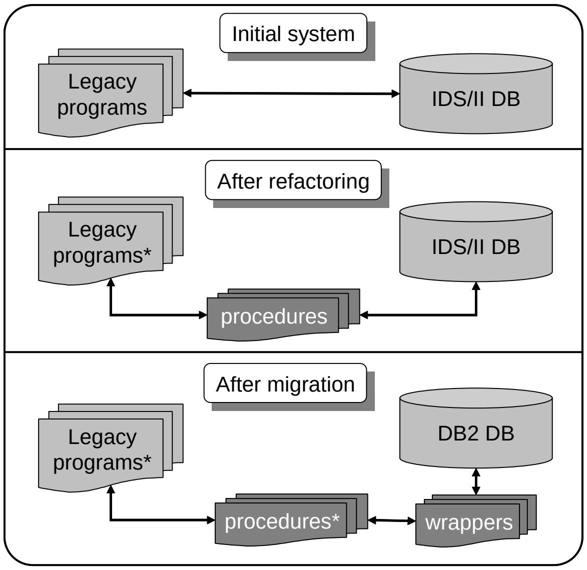


Figure 9.2: Project 2: refined methodology.

9.2 Project 2: IDS/II to DB2 with a refined methodology

Convinced by the results obtained during the first project, the customer asked us to carry out a second migration project. The goal of this second project was to migrate another CODASYL database (IDS/II) towards a relational platform (DB2). The legacy COBOL system also runs on a Bull GCOS8 mainframe and is made of about 1000 programs, totaling more than one million lines of code. Similarly to the first project, the target system must consist of the same COBOL programs running on the mainframe but remotely accessing a DB2 database through a DBSP gateway.

The methodology used in this second project slightly differs from the first one. The migration process comprised two main stages, as shown in Figure 9.2. The first phase involved the refactoring of the legacy application programs, while the second phase aimed at migrating the system towards the relational database platform. Both steps are summarized below.

9.2.1 System refactoring

The initial *system refactoring* stage consisted in centralizing the data manipulation statements within additional procedures. Each procedure was dedicated to a particular database operation applied to a particular record type. These procedures were generated from the legacy database schema. The application programs were then automatically transformed in such a way that each database operation was replaced with the corresponding procedure call.

	Transformed	Manually adapted	Source code size (kLOC)
# programs	996	10	800
# copybooks	300	0	43
# IDS/II inst.	22 395	15	-

Table 9.3: Project 2: Legacy program refactoring results

The main purpose of the refactoring step was to facilitate the migration phase itself by isolating the data manipulation primitives from the legacy source code. Indeed, no further alteration of the refactored legacy programs was necessary at the time of actually migrating the database.

The system refactoring step was supported by two distinct tools. The first one is a plugin of DB-MAIN (DB-MAIN, 2006) allowing to generate the additional procedures from the source IDS/II schema. The generated procedures are COBOL code sections distributed in a set of COBOL copybooks. Each copybook corresponds to an independent subset of the legacy schema.

The second tool, presented in Chapter 8, supports the transformation of the legacy programs. The source code transformation involved (1) the replacement of each IDS/II primitive with a procedure call, (2) the insertion of new variable declarations and (3) the introduction of the generated procedures by means of `COPY` statements. The program transformation tool takes as inputs (1) a COBOL program, (2) specific information on the legacy schema and (3) naming conventions related to the additional procedures.

Table 9.3 summarizes the results obtained during source code refactoring. A total of 22 395 IDS/II statements have been rewritten as procedure calls (`PERFORM` statements). We notice that 99% of the legacy programs were fully transformed without manual intervention. The automated transformation of the programs and copybooks took around 90 minutes[2]. Manual transformations were necessary in the presence of some variants of the `FIND` statement. For instance, as seen in Chapter 8, a record can be retrieved based on its technical identifier (`DB-KEY`) that is *global* to the entire database. It is not always possible to identify the record type accessed by such a statement through static program analysis. For instance, the statement `FIND DB-KEY IS` *`var`* identifies the record whose database key value is equal to the value of variable *`var`*. The type of the identified record thus depends on the value of *`var`*, which is rarely possible to determine through source code analysis.

The generated procedures consisted of around 42 000 lines of code. We observed an increase of the legacy source code size of about 4%. This small expansion is mainly due to the fact that each rewritten IDS/II statement is kept as a comment in the target programs.

[2] In this second project, the program transformation tool was executed on a more powerful machine than the one used in the first project.

	Physical schema (IDS/II)	Logical schema (IDS/II)	Conceptual schema	Relational schema (DB2)
# entity types	120	112	105	148
# rel. types	73	68	110	0
# attributes	1 283	1 509	1 204	1 841
max. # att/ent.type	43	42	35	56

Table 9.4: Project 2: Comparison of successive versions of the database schema

9.2.2 System migration

In the (*system migration*) process, the legacy database was migrated towards the DB2 platform. This was done in two phases, as depicted in Figure 9.3. Phase I was a transition phase, during which both the legacy and the target databases were maintained. The generated procedures were used to record each modification of the IDS/II database in a log file. On a daily basis, this log file served as input for replicating the modifications on top of the target DB2 database, allowing both databases to remain synchronized. During phase I, new applications were developped and tested on top of the target DB2 database.

During the second phase, database wrappers were generated from the source-to-target schema mapping. The additional procedures were then re-generated in order to make them access the target database through the generated wrappers.

Schema conversion based on database reverse engineering

The schema conversion results obtained in this project are summarized in Table 9.4. For each schema, the table gives the number of (1) entity types (record types or tables), (2) relationship types (set types or foreign keys) and (3) attributes (fields or columns). Multiple implicit constructs were discovered including 76 foreign keys, finer-grained attribute decompositions, as well as attribute format and cardinality constraints. The refined schema contains a total of 1509 fields among which 655 revealed to be optional !

Data analysis

A systematic data analysis phase was necessary before the data migration process could be performed. Indeed, the data must comply with the DDL[3] constraints of the target database in order to be succesfully migrated. In this project, we particularly focused on implicit foreign keys and format constraints. The data analysis process involved the following steps

1. The legacy IDS/II data were loaded into an intermediate DB2 database, in which each column was defined of type string, allowing error-free data

[3] DDL stands for Data Definition Language.

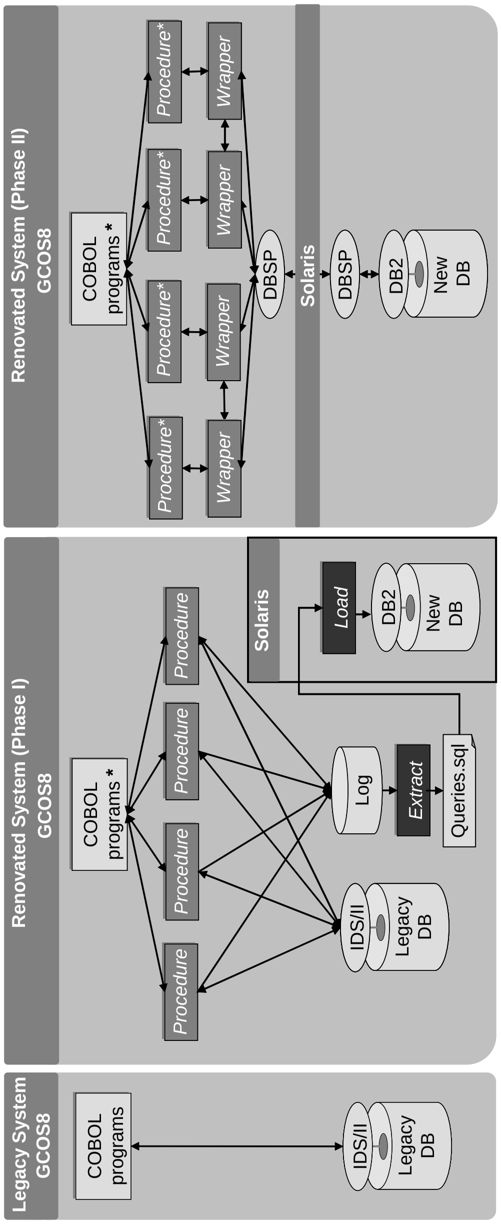


Figure 9.3: Project 2: Two-phase system migration.

migration. The resulting DB2 database contained more than 415 millions of rows.

2. The underlying database schema was then enriched with the following annotations:
   - The expected format of each DB2 column;
   - The validity constraints associated to each column (expressed as SQL *where* clauses);
   - The implicit foreign keys recovered during the reverse engineering phase;
3. Starting from this annotated schema, a data analysis program generated a set of SQL queries for inspecting the intermediate DB2 database. This analysis produced an html document reporting on the detected data errors.

The data analysis tool considered 76 implicit foreign keys, among which 24 proved to be violated by the legacy data. Regarding data format, a total of 3497 SQL queries were executed, allowing the detection of data inconsistencies in almost 70% of the record types.

A typical problem regarding format constraints was the presence of invalid dates. In the IDS/II database, date fields were represented as numeric values of the form ‘YYYYMMDD‘. In the target DB2 database, they are now expressed as columns of SQL type `Date`. We encountered a significant amount of inconsistent dates like ’20070931’ (31th of September). These errors were due to the behavior of some application programs considering day ’31’ as *the last day of the current month*, whatever the month. We also identified special date values like ’00000000’ or ’99999999’ that were used by programs for simulating the *null* value and a future date, respectively.

For several reasons, all the detected data inconsistencies could not be corrected by the customer, which obliged us to relax some constraints. For instance, some recovered foreign keys have not been explicitly declared in the target DDL code. Instead, fault-tolerant triggers were defined that produce a warning in a dedicated log table each time the referential constraint is violated by a database operation. This approach, which temporarily *tolerates* inconsistency (Balzer, 1991), allowed to improve referential integrity management, without being obliged to correct all the data nor to modify all the legacy programs.

Data migration

The data migration itself could also be automated, based on the mapping holding between the source and target database schemas. According to our approach, such a mapping is maintained and propagated while successive transformations are applied to the source schema. Thus both the IDS/II and the DB2 schemas were annotated with correspondence information which allows us to automatically generate the data migration program.

Wrapper generation and interfacing

Similarly, the source code of the wrappers was derived from the source-to-target schema mapping by a dedicated DB-MAIN plugin. The generated wrappers simulate each legacy DML statement on top of the target database. In the context of the present project, the wrappers translate IDS/II primitives using Embedded SQL code fragments.

Interfacing the generated wrappers with the legacy code was simply done by replacing the procedures introduced during the system refactoring phase with new, automatically generated procedures. Instead of accessing the IDS/II database, those new procedures now invoke the wrappers in order to access the DB2 database.

The target system is currently in use in the Belgian ministry on a day-to-day basis. The overall project was considered as a success by the customer.

9.3 Evaluation

Target database quality The database conversion approach we used, based on an initial database reverse engineering process, has the merit of producing a high-quality, fully-documented target database. In both projects, the DB2 database was designed as a native, normalized relational database, which does not look like the network database it was derived from. This means that new applications (in this particular case, web applications) can now directly access the DB2 database without invoking the wrappers. In addition, this approach will allow to smoothly migrate or rewrite the legacy code, until the wrappers become useless.

Flexibility The program conversion methodology used in the second project permits much more flexibility in the migration process itself. The main advantage is that the transformation of the legacy code is decoupled from the migration of the database. Recompiling and testing the automatic transformation of the programs can be done very early in the migration process. Furthermore, once the additional procedures (copybooks) have been introduced, the database can be *incrementally* migrated towards the target platform (subset by subset), while iteratively replacing the corresponding procedures. Finally, the legacy programs refactoring preserves the readability of the target source code. The developers of the legacy application still recognize their programs and can maintain them more easily. The only difference is that each IDS/II instruction is now written as a procedure call. The logic of the legacy programs remains unchanged. The new database (and its new paradigm) is fully hidden behind the generated wrappers.

Correctness In both projects, a systematic testing phase has shown the correctness of the program conversion step. This testing process was done in collaboration with IDS/II experts from the customer side. Two kinds of tests have been conducted. First, dedicated testing programs were written and executed, in order to verify that each kind of IDS/II statement was correctly simulated by the generated wrappers. This verification paid a particular attention to the management of both currency and exception status indicators. Second, a non-regression test was performed, during which the behaviour of the programs before and after their migration were systematically compared (based on reports produced and database state).

Performance The main drawback of our system conversion approach concerns performance. Indeed, we observed a significant performance degradation in both projects, the execution of some programs becoming up to 5 times slower than before. We explain the degradation by several factors among which (1) the database technology change, (2) the introduction of the wrapper layer and (3) the remote access to the target database through DBSP. As expected, the level of performance degration proved to be largely dependent on the data manipulation logic of the legacy programs. In case some programs become too slow, it is possible to rewrite them such that they *directly* access the migrated relational database in a more *natural way*. In the first project, some sequential reading loops were replaced with native random accesses, by including the searching criteria in the *where* clause. This allowed the adapted programs to reach the same level of performance as the original programs.

9.4 Conclusions and lessons learned

This chapter illustrated the reality of large-scale database migration projects, and showed that such projects may greatly benefit from a systematic, tool-supported methodology. The migration approach we used relies on a balanced combination of analysis, generative and transformational techniques. It allowed us to reach a good tradeoff between a high-level of automation of the migration process, and the maintainability of the target database and programs. Below, we briefly elaborate on the major lessons we learned from the two migration projects.

Full automation is not realistic Large-scale database migration projects obviously calls for scalable tool support. Although the above projects show that a high level of automation can be reached, fully-automating the process is clearly unrealistic. Indeed, such projects generally require several iterations and involve multiple human decisions.

Forward engineering is not straightforward Deriving a relational DDL code from a conceptual database schema can theoretically be performed automatically. In practice, when dealing with real schemas, external constraints must be taken into account during the schema design process. In this project, several decisions were guided by customer preferences like column type selection, naming conventions, and conversion strategies for compound and multivalued fields.

Data analysis is essential While data reverse engineering allows to discover implicit schema constraints, making all these rules explicit in the target schema is not always achievable. We observed, in both projects, that a significant subset of the legacy data instances actually violate the implicit rules recovered by program inspection. In addition, We also learned that database reverse engineering may greatly benefit from data analysis. Indeed, analyzing database contents does not only allow to detect errors or to validate integrity rules. It may also serve as a basis for formulating *new* hypotheses about potential implicit constraints. The main limitation is that, as just explained, we cannot assume that the legacy database is in a consistent state.

Wrapper development is challenging Developing correct wrappers requires a precise knowledge of the legacy data manipulation system. In both projects, the task was challenging due to the paradigm shift between CODASYL and relational database systems. Indeed, the generated wrappers must precisely simulate the behaviour of the IDS/II primitives, which includes the management of currency indicators, reading sequences and returning status codes (as seen in Chapter 8). Another challenge, as for the data migrators, was to correctly deal with IDS/II record types that had been split into several tables.

Roadmap

This chapter concludes the part dedicated to the automated adaptation of programs under database platform migration. This part mainly focused on the *language consistency* problem of database evolution, by elaborating on how to translate queries expressed in the source query language into equivalent queries expressed in the target query language. In the next part (Part V), we will concentrate on the *structural consistency* relationship that must hold between the application programs and the schema of their database. Chapter 10 will present a co-transformational approach to database schema refactoring, according to which semantics-preserving schema transformations are associated to program transformations. The latter aim to convert the database queries expressed on the source schema into equivalent queries expressed on the target schema.

Part V

Adapting Programs to Database Schema Change

Chapter 10

A Co-transformational Approach to Schema Refactoring

Each problem that I solved became a rule, which served afterwards to solve other problems.[1]
– Rene Descartes

This chapter[2] addresses the problem of consistency preservation in data-intensive system evolution. When the database structure evolves, the application programs must be changed to interface with the new schema. The latter modification can prove very complex, error prone and time consuming. We describe a comprehensive co-transformational approach according to which automated program transformations can be derived from schema transformations.

10.1 Introduction

Software evolution consists in keeping a software system up-to-date and responsive to ever changing business and technological requirements. This chapter focuses on the evolution of complex database applications, that is, data-intensive software systems comprising a database. Database migration, database merging and database restructuring are popular evolution scenarios that involve not only changing the data components of applications, but also rewriting some parts of the programs themselves, even when no functional change occurs. In general, such evolution patterns induce the modification of three mutually dependent system components, namely the data structures (i.e., the *schema*), the data instances and the application programs (Hick and Hainaut, 2006). When the system evolves, the consistency that exists between these three artefacts must be preserved.

[1] Citation translated from French

[2] An earlier version of this chapter (Cleve and Hainaut, 2006) appeared in the tutorial book *Generative and Transformational Techniques in Software Engineering*, published as volume 4143 of *Lecture Notes in Computer Science*, Springer, 2006.

In this chapter, we focus on the consistency relationship that holds between the application programs and their database schema. We assume that the evolution process starts with a schema modification, potentially challenging this consistency. Our main question is the following: *how can a change in a database schema be propagated to the application programs manipulating its data instances*?

Based on the observation that (1) any schema change can be modelled by a *schema transformation*, and (2) any program modification can be carried out using *program transformation* rules, this chapter elaborates on the possible coupling of schema transformations and program transformations for supporting the co-evolution of database schema and associated programs.

The chapter is structured as follows. Section 10.2 briefly summarizes our general approach. Section 10.3 presents the LDA language on top of which the program transformation rules of this chapter are defined. Section 10.4 recalls the concept of schema transformation. The way program transformations can be derived from schema transformations is discussed in Section 10.5. Section 10.6 systematically associate a set of abstract program transformation rules to standard schema transformations. In Section 10.7, we illustrate the application of our general approach in three particular evolution contexts, namely schema refactoring, database design and database migration. Section 10.8 gives an overview of a tool architecture that support the whole process. We discuss related work in Section 10.9, while Section 10.10 further clarifies the actual objectives and contributions of the chapter. Concluding remarks are given in Section 10.11.

10.2 General approach

The general approach presented in this chapter is summarized in Figure 10.1. For the sake of genericity, we will consider (1) the GER data model as a generic model for database schemas and (2) the LDA language as an abstract data manipulation language. Each representative GER-to-GER schema transformation will be associated with LDA program transformations allowing to preserve the structural consistency that must hold between the programs and the evolving schema.

The proposed approach considers (1) a *limited subset* of the GER model constructs and (2) a *limited subset* of the schema transformations that can be applied to those constructs:

- The GER constructs not considered are is-a hierarchies, relationship types with attributes, many-to-many relationship types and n-ary relationship types (with n > 2).
- The GER-to-GER transformations not considered are (1) those that are not semantics-preserving and (2) those that produce or apply to GER constructs not considered.

The subset of GER constructs considered still encompasses the major existing data models including the relational model, the COBOL file model, and the CODASYL

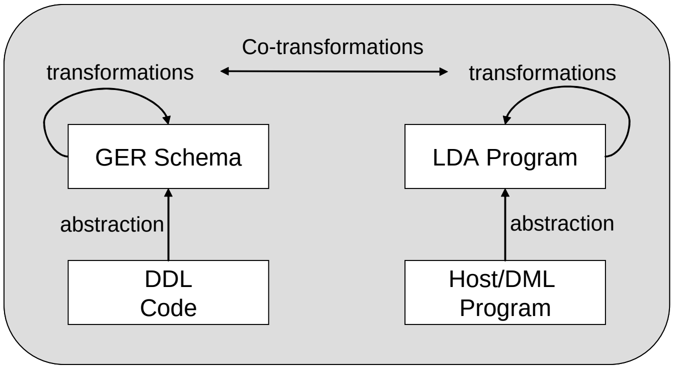


Figure 10.1: Co-transformational approach based on the GER model and the LDA language.

model. Consequently, the abstract co-transformation rules we will define below can be instantiated in the context of relational schema refactoring, COBOL file structures refactoring or CODASYL schema refactoring. Similarly, the same co-transformation rules can serve as a basis for supporting the conversion of a COBOL or a CODASYL schema into a relational schema.

10.3 The LDA language

The LDA language is a semi-procedural language including database manipulation primitives. A LDA program is associated to a GER schema G. It may manipulate the instances of any object specified in G via host variables of corresponding types. The navigation through the database always starts from an entity type.

LDA has been adapted from a similar language defined by Hainaut (1986). It now combines usual language constructs of a query language and a programming language, among which:

- Data manipulation primitives for selecting, creating, deleting and modifying database records[3];
- Types: integers, strings, booleans, and *GER types*[4]
- Conditional statements: *If-then-else*, *For-loops*, *While-loops*
- I/O statements: `input`, `print`

[3] Here, *record* means *entity type instance*.

[4] i.e., references to possibly complex data types specified in the underlying GER schema.

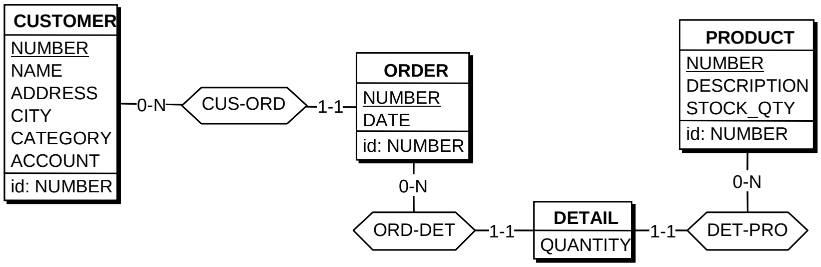


Figure 10.2: Sample GER schema.

10.3.1 Database manipulation in LDA

In this section, we briefly describe the main features of the LDA language as far as database manipulation is concerned. Our examples will be based on the sample GER schema of Figure 10.2, describing customers, orders, order details and products.

Record selection expression

A record selection expression has the following general form:

entityTypeName [*selectionCondition*]

where:

- *entityTypeName* is the name of an entity type of the underlying schema;
- *selectionCondition* expresses the condition under which a record is selected.

The result of the evaluation of such an expression consists of a *set* of records of type *entityTypeName* that satisfy *selectionCondition*. Some examples of record selection expressions, defined on top of the schema of Figure 10.2, are given below:

- *the set of customers...*
  `CUSTOMER`
- *the set of customers living in Namur...*
  `CUSTOMER (:CITY = 'Namur')`
- *the set of orders placed by a customer who lives in Namur...*
  `ORDER(CUS-ORD:CUSTOMER(:CITY = 'Namur'))`
- *the set of products for which the stock quantity is greater than 10...*
  `PRODUCT(:STOCK-QTY > 10)`

Assignment of record selection expression

A record selection expression can be assigned to a variable, as illustrated below:

```
  cus := CUSTOMER(:CITY = 'Namur');
```

If the set resulting from the evaluation of the record selection expression:

- is empty, the variable is set to *null*.
- contains only one element, the variable refers to this element for subsequent statements.
- contains more than one element, one record of the set is randomly chosen and its reference is assigned to the variable.

Typically, such assignments are used when the record selection expression returns a singleton (i.e., one record).

Conditional statement based on record selection expression

A record selection expression used as a condition evaluates to true if the corresponding set of records is non-empty. For instance, the following `if` statement displays 'yes' if at least one customer lives in Namur, and displays 'no' otherwise.

```
if (CUSTOMER(:CITY = 'Namur'))
  then print('yes')
  else print('no')
endif
```

The above code fragment is equivalent to:

```
cus := (CUSTOMER(:CITY = 'Namur'))
if (cus = null)
  then print('yes')
  else print('no')
endif
```

For loops based on record selection expression

When the program needs to iterate on a set of records, it makes use of a *for loop*. As an example, the following code fragment displays the name of each customer living in Namur.

```
for cus := CUSTOMER(:CITY = 'Namur') do
    print(cus.NAME)
endfor;
```

In this case, the record selection expression is evaluated *before* the first iteration. The body of the loop is executed n times, where n is the number of elements of the resulting set of records. At each iteration, the loop variable (here, `cus`) references a different record of this set.

LDA	COBOL	CODASYL	SQL
`create`	`WRITE`	`STORE`	`INSERT`
`delete`	`DELETE`	`ERASE`	`DELETE`
`update`	`REWRITE`	`MODIFY`	`UPDATE`

Figure 10.3: Approximate correspondences between data modification primitives.

Data modification primitives

The LDA language provides the following usual data modification primitives allowing to create, delete and update database records:

- `create` *var* `:=` *entityTypeName condition*: creates a record of type *entityTypeName* satisfying *condition*. Variable *var* references the created record for further manipulation.
- `delete` *var* `[`*condition*`]`: deletes the record referenced by *var* if it satisfies *condition*. If the record is deleted, variable *var* is set to *null*.
- `update` *var condition*: updates the record referenced by *var* such that it satisfies *condition*.

Figure 10.3 presents the approximate correspondences between the above abstract LDA primitives and concrete DML statements in COBOL, CODASYL and SQL. Access (reading) primitives are both more simple and more complex than modification primitives. In the one hand, the instance mapping states how instances can be extracted from the database according to the new schema. On the other hand, the way *currency registers* are implemented in various DMS can be quite different[5]. Abstracting them in a DMS-independent manner is too complex to be addressed in this chapter. Therefore, we assume, without loss of generality, that propagating schema transformations to *reading* primitives requires DMS-specific rules. As far as reading database access is concerned, we will only consider the *record selection expressions* presented above, that allow to select one or more instances of a given entity type based on (1) the value of their attributes and (2) their relationships with other records.

10.3.2 Illustration

Figure 10.4 shows a sample LDA program allowing the creation of a new instance of entity type `ORDER` of Figure 10.2. The head of an LDA program comprises (1) the name of the program; (2) a reference to the associated GER schema; and (3) program variable declarations. As previously mentionned, the type of a LDA variable can be either a simple type (integer, boolean, string,..), or a GER type. For instance, the type of variable `prod` and `prod-code` reference entity type `PRODUCT` and attribute `PRODUCT.NUMBER`, respectively. Regarding the source code body itself,

[5] As it can be observed in Chapters 7 and 8 of this thesis.

the program first accepts the identification number of the new order together with its date, then it verifies that the provided customer number exists and creates the order itself. Finally, it creates one detail per ordered product (the specified product number should also exist).

10.4 Schema transformations

A schema transformation consists in deriving a target schema S' from a source schema S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty) (Hainaut, 2006). C (resp. C') is empty when the transformation consists in adding (resp. removing) a construct. A more formal definition is provided below:

Definition 3 *A schema transformation Σ is a couple of mappings $\langle T, t\rangle$ such that : $C' = T(C)$ and $c' = t(c)$, where c is any instance of C and c' the corresponding instance of C'.* Structural mapping *T explains how to modify the schema while* instance mapping *t states how to compute the instance set of C' from instances of C.*

Semantics preservation Any transformation Σ can be given an inverse transformation $\Sigma^{-1} = \langle T^{-1}, t^{-1}\rangle$ such that $T^{-1}(T(C)) = C$. If, in addition, there also exists an instance mapping t^{-1} such that: $t^{-1}(t(c)) = c$, then Σ (and Σ^{-1}) are said *semantics-preserving* or *reversible.* If $\langle T^{-1}, t^{-1}\rangle$ is also reversible, Σ and Σ^{-1} are called *symmetrically reversible.* If a schema transformation is reversible, then the source schema can be replaced with the target one without loss of information. We refer to (Hainaut, 2006) for a more detailled analysis of semantics preservation in schema transformations.

As already identified in Chapter 3, three schema transformation categories exist: (1) semantics-*augmenting* schema transformations (S^+), (2) semantics-*decreasing* schema transformations (S^-) and (3) semantics-*preserving* schema transformations ($S^=$).

Examples Figure 10.5 graphically illustrates the structural mapping T_1 of the transformation of a compound attribute into an entity type and a relationship type R. Figure 10.6 depicts the structural mapping T_2 of the transformation of a one-to-one relationship type R into a foreign key. Both transformations can be proved to be semantics-preserving.

10.5 Program adaptation by co-transformations

The feasibility of automatically adapting a program to a schema transformation depends on the nature of the latter. In the general case, the transformations of S^+ and S^- categories do not allow automatic program modifications. The task remains under the responsibility of the programmer. However, it is generally possible to

```
// program identification
program NEW_ORDER.

// reference to the underlying GER schema
schema ’create_ord.lun’;

// declaration of program variables
ord : ORDER;
ord-num : ORDER.NUMBER;
ord-date : ORDER.DATE;
cus : CUSTOMER;
cus-num : CUSTOMER.NUMBER;
det : DETAIL;
ord-qty : DETAIL.QUANTITY;
prod : PRODUCT;
prod-code : PRODUCT.NUMBER;
is-ok, stop : boolean;
integer : i;


begin
  print(’Creating a new order...’);
  print(’Enter order number:’);
  input(ord-num);
  print(’Enter order date:’);
  input(ord-date);
  is-ok := false;
  // verification loop: the specified customer should exist
  while (is-ok = false) do
      print(’Enter customer id:’);
      input(cus-num);
      if (CUSTOMER (:NUMBER = cus-num))
          then is-ok := true;
               cus := CUSTOMER (:NUMBER = cus-num)
          else print(’Unknown customer !’)
      endif
  endwhile;
  // creation of the new order
  create ord := ORDER ((:NUMBER = ord-num)
                   and (:DATE = ord-date)
                   and (CUS-ORD : cus));
  print(’Order ’, ord.NUMBER, ’ created for customer ’, cus.NUMBER);
  stop := false;
  // creation of one order detail per ordered product
  while (stop = false) do
      print(’Add order detail? (yes=1, no<>1):’);
      input(i);
      if (i = 1)
          then  is-ok := false;
                // verification loop: the specified product should exist
                while (is-ok = false) do
                    print(’Enter product code:’);
                    input(prod-code);
                    if (PRODUCT (:NUMBER = prod-code))
                        then  is-ok := true;
                              prod := PRODUCT (:NUMBER = prod-code)
                        else  print(’Unknown product!’)
                    endif
                endwhile;
                print(’Enter quantity ordered:’);
                input(ord-qty);
                create det := DETAIL((:QUANTITY = ord-qty)
                                 and (ORD-DET : ord)
                                 and (DET-PRO : prod))
          else  stop := true
      endif
  endwhile
end.
```

Figure 10.4: Example LDA program allowing the creation of an ORDER.

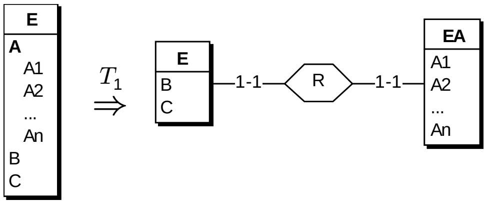


Figure 10.5: Structural mapping T_1 of a semantics-preserving schema transformation that transforms a compound attribute A into entity type EA and a relationship type R. (instance representation)

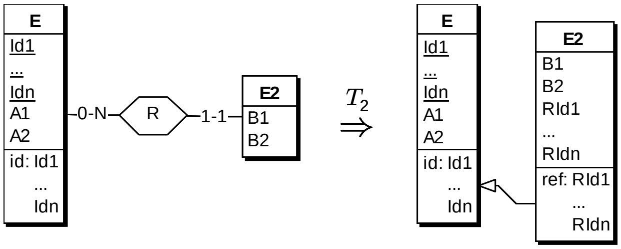


Figure 10.6: Structural mapping T_2 of a semantics-preserving schema transformation that transforms a one-to-many relationship type R into a foreign key $RId_1 \cdots RId_n$.

evaluate the impact of such semantic evolutions by automatically locating the program sections where occurrences of modified object types are processed (Henrard, 2003).

In contrast, semantics-preserving schema transformations ($S^{=}$) can be propagated to the program level more easily. Indeed, they allow the programming logic to be left unchanged, since the application programs still manipulate the same informational content. Program conversion mainly consists in adapting the related DMS[6] statements to the modified data structure.

Co-transformational approach

When modelling schema modifications as schema transformations, the program adaptation problem translates as follows. *Given a semantics-preserving schema transformation Σ applicable to data construct C, how can it be propagated to the database queries that select, create, delete and update instances of construct C?* Our approach consists in associating with structural mapping T of Σ, in addition to instance mapping t, a query rewriting mapping stating how to adapt the related queries accordingly. In other words, we propose to extend the concept of *schema transformation* to the more general term of *database co-transformation*, as formally defined below.

Notations The following notations will be used:

- $\mathcal{S}$ denotes the set of all possible database schemas.
- $\mathcal{D}(S)$ denotes the set of all possible database states complying with schema $S \in \mathcal{S}$.
- $\mathcal{Q}(S)$ denotes the set of all possible queries that can be expressed on schema $S \in \mathcal{S}$.
- $\mathcal{R}(S)$ denotes the set of all possible results of reading queries on database states complying with schema S.

Definition 4 *A database query q expressed on a schema S is a function $q : \mathcal{D}(S) \rightarrow \mathcal{D}(S) \times \mathcal{R}(S)$, that takes a database state $d \in \mathcal{D}(S)$ as input, and returns a (possibly updated) database state $d' \in \mathcal{D}(S)$ together with a (possibly empty) result $r \in \mathcal{R}(S)$.*

Reading primitives typically leave the database state unchanged, in contrast with modification primitives (create, delete, update) that affect the database contents.

Definition 5 *A database co-transformation $\Phi = \langle T, t_d, t_q \rangle$ is a 3-tuple of transformations where:*

- *$T : \mathcal{S} \rightarrow \mathcal{S}$ transforms the schema;*

[6] Data Management System

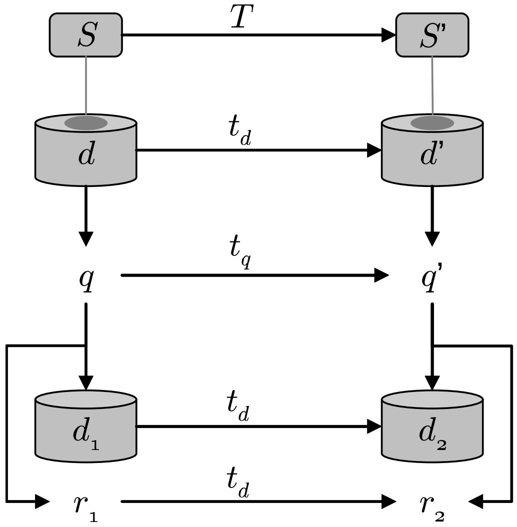


Figure 10.7: Graphical representation of a database co-transformation.

- $t_d : \mathcal{D}(S) \rightarrow \mathcal{D}(T(S))$ *transforms the data instances;*
- $t_q : (\mathcal{D}(S) \rightarrow \langle\mathcal{D}(S) \times \mathcal{R}(S)) \rightarrow (\mathcal{D}(T(S)) \rightarrow \mathcal{D}(T(S)) \times \mathcal{R}(T(S)))$ *transforms the database queries such that* $\forall d \in \mathcal{D}(S) : \forall q \in \mathcal{Q}(S) :$
  - $t_q(q) \in \mathcal{Q}(T(S))$
  - *let* $q(d) = \langle d_1, r_1\rangle \wedge t_q(q)(t(d)) = \langle d_2, r_2\rangle$
    then $d_2 = t_d(d_1) \wedge r_2 = t_d(r_1)$ *and, conversely,* $d_1 = t_d^{-1}(d_2) \wedge r_1 = t_d^{-1}(r_2)$.

Figure 10.7 graphically summarizes the above definitions.

Illustrations Figure 10.8 illustrates the query transformation mapping t_{q_1} that can be associated with structural mapping T_1 of Figure 10.5. Since attribute A of entity type E has become entity type EA, the way of creating an instance of E must be adapted accordingly. It now involves the creation of an instance of entity type EA corresponding to the old A instance. The created EA instance must be linked with the instance (e) of E through relationship type R.

Figure 10.9 illustrates the query transformation mapping t_{q_2} associated with structural mapping T_2 of Figure 10.6. The relationship condition $R : e$ of the create primitive is now expressed as an equality condition between the foreign key value and the target identifier value.

```
create e := E(:A.A_1 = a_1                 create e := E(:B = b and :C = c);
              ...                   t_q1   create ea := EA(:A_1 = a_1
              and :A.A_n = a_n       =>                  ...
              and :B = b                                 and :A_n = a_n
              and :C = c)                                and R:e)
```

Figure 10.8: Query transformation mapping t_{q_1} associated with structural mapping T_1 of Figure 10.5, when applied to a create primitive.

```
create e2 := E2(:B_1 = b_1                create e2 := E2(:B_1 = b_1
                and :B_2 = b_2     t_q2                   and :B_2 = b_2
                and R :  e)         =>                    and :RId_1 = e.Id_1
                                                             ...
                                                          and :RId_n = e.Id_n)
```

Figure 10.9: Query transformation mapping t_{q_2} associated with structural mapping T_2 of Figure 10.6, when applied to a create primitive.

10.6 Co-transformation rules

In this section, we specify a set of co-transformation rules, according to which a set[7] of semantics-preserving GER-to-GER schema transformations are associated with corresponding program transformation rules defined on the LDA language. For each specified co-transformation $\Phi = \langle T, t_d, t_q \rangle$, we successively present:

- the signature of T;
- the graphical representation of T;
- the program transformation rules (t_q) associated to T.
- the intuitive justification of the correctness of those program transformation rules.

Assumptions

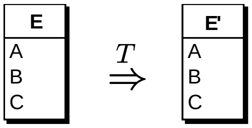
- The initial program is assumed (1) to be correct, (2) to comply with the initial database schema and (3) to respect to underlying data integrity constraints.
- Each program transformation rule assumes that its right-hand side constitutes an *indivisible* sequence of primitives, similarly to a code fragment falling within the scope of a database *transaction*. This assumption is required since some of the program transformation rules replace a *single* instruction by a *sequence* of instructions.

[7] The co-transformation rules do not claim to cover all possible semantics-preserving transformations, but rather consider the most representative ones.

10.6.1 Entity type renaming

Signature $E' \leftarrow \mathit{Rename}\text{-}\mathit{ET}(E)$

Structural mapping The *Rename-ET* transformation changes the name of an entity type E as E'.



Program transformation The adaptation of the queries is straightforward. It consists in replacing each reference to entity type E with a reference to E'.

- variable declaration :
  `e :` E`;` $\Rightarrow$ `e :` E'`;`

- record selection expressions:
  E`(`$cond$`)` $\Rightarrow$ E'`(`$cond$`)`

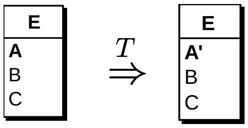
- create primitives:
  `create e :=` E`(`$cond$`)` $\Rightarrow$ `create e :=` E'`(`$cond$`)`

Correctness Since the *Rename-ET* transformation does not involve any modification of the data instances, the correctness proof of the associated program transformation rules is immediate.

10.6.2 Attribute renaming

Signature $A' \leftarrow \mathit{Rename}\text{-}\mathit{ATT}(E, A)$

Structural mapping The *Rename-ATT* transformation gives a new name A' to an attribute A of an entity type E.



Program transformation Each reference to attribute $E.A$ must then be replaced with a reference to attribute $E.A'$.

- variable declaration :
  `a :` $E.A$`;` $\Rightarrow$ `a :` $E.A'$`;`

- record selection expressions:
  $E(\ldots : A\ \ rel\ \ exp \ldots) \Rightarrow E(\ldots : A'\ \ rel\ \ exp \ldots)$

- create primitives:
  `create e :=` $E(\ldots : A$ `=` $exp \ldots) \Rightarrow$ `create e :=` $E(\ldots : A'$ `=` $exp \ldots)$

- delete primitives:
  `delete e(`$\ldots : A\ \ rel\ \ exp \ldots$`)` $\Rightarrow$ `delete e(`$\ldots : A'\ \ rel\ \ exp \ldots$`)`
  where type(`e`) $= E$

- update primitives:
  `update e(`$\ldots : A$ `=` $exp \ldots$`)` $\Rightarrow$ `update e(`$\ldots : A'$ `=` $exp \ldots$`)`
  where type(`e`) $= E$

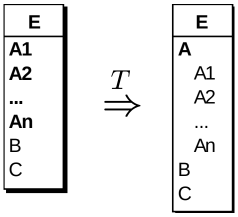
- attribute reference:
  `e.`$A \Rightarrow$ `e.`A'
  where type(`e`) $= E$

Correctness Similarly to the previous transformation, *Rename-ATT* does not necessitate the alteration of the data instances. The program transformation rules simply consists in changing the name of the attribute accordingly.

10.6.3 Attribute aggregation

Signature $A \leftarrow Aggregate\text{-}ATT(E, \{A_1, A_2, ..., A_n\})$

Structural mapping The *Aggregate-ATT* transformation groups a set of first-level attributes $\{A_1, A_2, ..., A_n\}$ of an entity type E within a single compound attribute A of the same entity type.



Program transformation Each reference to first-level attribute $E.A_i$ must be rewritten as a reference to the corresponding sub-level attribute of A.
$\forall i : 1 <= i <= n :$

- variable declaration :
  $\mathtt{a}_i$ `:` $E.A_i$`;` $\Rightarrow$ $\mathtt{a}_i$ `:` $E.A.A'_i$`;`

- record selection expressions:
  $E(\ldots : A_i\ \ rel\ \ exp \ldots) \Rightarrow E(\ldots : A.A_i\ \ rel\ \ exp \ldots)$

- create primitives:
  create e := E(...:A_i = *exp*...) $\Rightarrow$ create e := E(...:$A.A_i$ = *exp*...)

- delete primitives:
  delete e(...:A_i *rel exp*...) $\Rightarrow$ delete e(...:$A.A_i$ *rel exp*...)
  where type(e) $= E$

- update primitives:
  update e(...:A_i = *exp*...) $\Rightarrow$ update e(...:$A.A_i$ = *exp*...)
  where type(e) $= E$

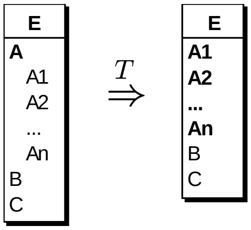
- attribute reference:
  e.A_i $\Rightarrow$ e.$A.A_i$
  where type(e) $= E$

Correctness The way the *Aggregate-ATT* transformation is propagated to the programs is easy to justify. This transformation involves, for each record, the aggregation of the values of simple attributes A_1, A_2, ..., A_n within a single compound attribute A, itself composed of sub-level attributes $A.A_1$, $A.A_2$, ..., $A.A_n$. A reference to a simple attribute A_i in the source schema translates into a reference to the corresponding sub-level attribute $A.A_i$ in the target schema.

10.6.4 Compound attribute disaggregation

Signature $\{A_1, A_2, ..., A_n\} \leftarrow \mathit{Disaggregate\text{-}ATT}(E, A)$

Structural mapping The *Disaggregate-ATT* transformation can be seen as the inverse of *Aggregate-ATT*. It discards a first-level compound attribute A, each of its sub-level attributes $\{A_1, A_2, ..., A_n\}$ becoming a first-level attribute.



Program transformation The program adaptation consists in replacing each reference to sub-level attribute $E.A.A_i$ with a reference to the corresponding first-level attribute $E.A_i$.
$\forall i : 1 <= i <= n$:

- variable declaration :
  a$_i$: $E.A.A_i$; $\Rightarrow$ a$_i$: $E.A_i$;

- record selection expressions:
  $E\texttt{(}\ldots\texttt{:}A.A_i \ \mathit{rel} \ \mathit{exp}\ldots\texttt{)} \Rightarrow E\texttt{(}\ldots\texttt{:}A_i \ \mathit{rel} \ \mathit{exp}\ldots\texttt{)}$

- create primitives:
  $\texttt{create e := } E\texttt{(}\ldots\texttt{:}A.A_i \texttt{ = } \mathit{exp}\ldots\texttt{)} \Rightarrow \texttt{create e := } E\texttt{(}\ldots\texttt{:}A_i \texttt{ = } \mathit{exp}\ldots\texttt{)}$

- delete primitives:
  $\texttt{delete e(}\ldots\texttt{:}A.A_i \ \mathit{rel} \ \mathit{exp}\ldots\texttt{)} \Rightarrow \texttt{delete e(}\ldots\texttt{:}A_i \ \mathit{rel} \ \mathit{exp}\ldots\texttt{)}$
  where $\mathit{type}(\texttt{e}) = E$

- update primitives:
  $\texttt{update e(}\ldots\texttt{:}A.A_i \texttt{ = } \mathit{exp}\ldots\texttt{)} \Rightarrow \texttt{update e(}\ldots\texttt{:}A_i \texttt{ = } \mathit{exp}\ldots\texttt{)}$
  where $\mathit{type}(\texttt{e}) = E$

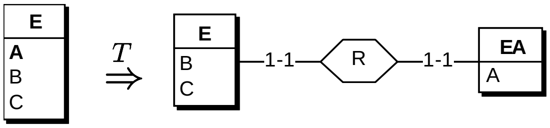
- attribute reference:
  $\texttt{e.}A.A_i \Rightarrow \texttt{e.}A_i$
  where $\mathit{type}(\texttt{e}) = E$

Correctness The justification of correctness for the *Disaggregate-ATT* is the exact converse of the one given for the *Aggregate-ATT* transformation.

10.6.5 Attribute to entity type (instance representation)

Signature $\langle R, EA\rangle \leftarrow \mathit{ATT\text{-}to\text{-}ET\text{-}inst}(E, A)$

Structural mapping The *ATT-to-ET-inst* transforms an attribute A into an entity type EA and a relationship type R, by *instance representation.* This means that there may exist multiple instances of entity type EA having the same value of attribute A, but each instance of EA is linked to *exactly one* instance of entity type E.



Program transformation The adaptation of the data manipulation primitives is as follows:

- variable declaration :
  $\texttt{a : } E.A\texttt{;} \Rightarrow \texttt{a : } EA.A\texttt{;}$

- record selection expressions:
  $E\texttt{(}\ldots\texttt{:}A \ \mathit{rel} \ \mathit{exp}\ldots\texttt{)} \Rightarrow E\texttt{(}\ldots R\texttt{:}EA\texttt{(:}A \ \mathit{rel} \ \mathit{exp}\texttt{)}\ldots\texttt{)}$

- create primitives: the creation of one instance of E now involves the creation of one instance of EA, linked to the created instance of E through relationship type R.

```
create e := $E$(:$A$ = $exp_1$ and :$B$ = $exp_2$ and :$C$ = $exp_3$)
$\Rightarrow$
create e := $E$(:$B$ = $exp_2$ and :$C$ = $exp_3$);
create ea(:$A$ = $exp_1$ and $R$:e)
```

- delete primitives: deleting an instance of E now implies to remove the associated instance of EA from the database.

```
delete e(:$A$ $rel$ $exp$) $where$ $type$(e) = $E$
$\Rightarrow$
ea = $EA$($R$:e);
if (ea.$A$ $rel$ $exp$) then
   delete ea;
   delete e
endif
```

- update primitives: updating the value of attribute A for an instance `e` of E now consists in updating the instance of EA associated to `e`.

```
update e(:$A$ = $exp$) $where$ $type$(e) = $E$
$\Rightarrow$
ea = $EA$($R$:e);
update ea(:$A$ = $exp$);
```

- attribute reference:

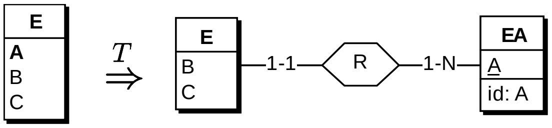
```
e.$A$ $\Rightarrow$ e.$R$.$A$
$where$ $type$(e) = $E$
```

Correctness The *ATT-to-ET-inst* transformation simply extracts an attribute A from an entity type E. For each record of type E, the value of attribute A is converted into a distinct record of type EA having the same value of attribute A. This created record is linked to its corresponding record of type E. Thus, referencing attribute A of an instance e of entity type E on the source schema, can be simulated through a reference to the value of attribute A for the record of type EA that is linked to e. Deleting a record e of type E on the target schema now necessitates the deletion of the instance of EA associated to e.

10.6.6 Attribute to entity type (value representation)

Signature $\langle R, EA \rangle \leftarrow ATT\text{-}to\text{-}ET\text{-}value(E, A)$

Structural mapping The *ATT-to-ET-value* transforms an attribute A into an entity type EA and a relationship type R, by *value representation*. In contrast with the *instance representation*, the same instance of entity type EA may be associated to multiple instances of entity type E, but there cannot exist two instances of EA having the same value of attribute A.



Program transformation The data manipulation primitives can be adapted as follows:

- variable declaration :
  `a :` $E.A$`;` ⇒ `a :` $EA.A$`;`

- record selection expressions:
  E`(...:`A *rel exp*`...)` ⇒ E`(...`R`:`EA`(:`A *rel exp*`)...)`

- create primitives: when creating a new instance of E, one must now check whether an instance `ea` of EA already exists with the proper value of attribute A. If it is the case, the created instance of E is linked to `ea` through relationship type R. If this is not the case, an instance of EA must be created beforehand.

```
create e := $E$(:$A$ = $exp_A$ and :$B$ = $exp_B$ and :$C$ = $exp_C$)
⇒
ea = $EA$(:$A$ = $exp_A$);
if (ea = null) then
   create ea := $EA$(:$A$ = $exp_A$)
endif;
create e := $E$($R$:ea and :$B$ = $exp_B$ and :$C$ = $exp_C$);
```

- delete primitives: deleting an instance of E now necessitates the deletion of the instance `ea` of EA associated to `e`. This is done only if `ea` is not also linked to other instances of E.

```
delete e(:$A$ $rel\ exp$)
$where\ type$(e) = $E$
⇒
ea = EA($R$:e);
if (ea.$A$ $rel\ exp$) then
   delete e;
   if not($E$($R$:ea)) then
      delete ea;
   endif
endif
```

- update primitives: updating the value of former attribute A for an instance `e` of E now consists in replacing the instance of EA associated to `e`. In case an instance `ea-new` of EA with the new value of A already exists, then this instance can be linked to `e`. Otherwise, a corresponding new instance of EA must be created. The instance `ea-old` of EA previously linked to `e` can be deleted if it is not linked to other instances of E.

```
update e(:$A$ = $exp$)
$where\ type$(e) = $E$
$\Rightarrow$
ea-old = $EA$($R$:e);
ea-new = $EA$(:$A$ = $exp$);
if (ea-new = null) then
   create ea-new := $EA$(:$A$ = $exp$)
endif;
update e($R$:ea-new);
if not($E$($R$:ea-old)) then
   delete ea-old
endif
```

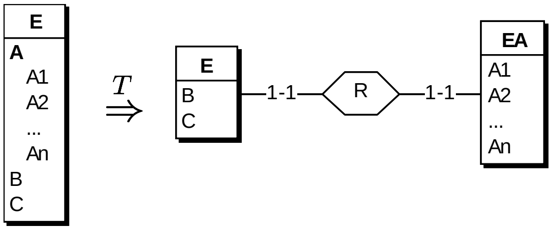
- attribute reference:
  `e.`A ⇒ `e.`R`.`A
  where type(`e`) $= E$

Correctness The justification of correctness is similar to the one of the *ATT-to-ET-inst* transformation except that, in this case, the same instance of EA can be linked to several instances of E. This means that when inserting an instance of E, one should first verify that an instance of EA with the desired value of attribute A does not already exist. Similar ad-hoc operations are also necessary (1) when updating the value of attribute A linked to a given instance of E and (2) when deleting an instance of E.

10.6.7 Compound attribute to entity type (instance representation)

Signature $\langle R, EA\rangle \leftarrow$ *CoATT-to-ET-inst*(E, A)

Structural mapping The *CoATT-to-ET-inst* transformation replaces a compound attribute A of an entity type E with an entity type EA having the same decomposition as A. The conversion is performed by *instance representation*, which means that each instance of new entity type EA corresponds to exactly *one* instance of entity type E through relationship type R. Duplicate values of attribute A among the instances of EA are allowed.



Program transformation Each reference to sub-level attribute $E.A.A_i$ must then be rewritten as a reference to the corresponding first-level attribute $EA.A_i$ $\forall i : 1 <= i <= n$:

- variable declaration :
  ```
  a : $E.A.A_i$; ⇒ a : $EA.A_i$;
  ```
- record selection expressions:
  ```
  $E$(...:$A.A_i$ $rel$ $exp$...) ⇒ $E$(...$R$:$EA$(:$A_i$ $rel$ $exp$)...)
  ```
- create primitives: the creation of one instance of E now involves the creation of one instance of EA, linked to the created instance of E through relationship type R.
  ```
  create e := $E$(:$A.A_1$ = $exp_{A_1}$ ... and :$A.A_n$ = $exp_{A_n}$ and :$B$ = $exp_2$
  and :$C$ = $exp_3$)
  ⇒
  create e := $E$(:$B$ = $exp_2$ and :$C$ = $exp_3$);
  create ea(:$A_1$ = $exp_{A_1}$ ... and $A_n$ = $exp_{A_n}$ and $R$:e)
  ```
- delete primitives: deleting an instance of E now implies to remove the associated instance of EA from the database.
  ```
  delete e($A.A_i$ $rel$ $exp$)
  where type(e) = $E$
  ⇒
  ea = $EA$($R$:e);
  if (ea.$A_i$ $rel$ $exp$) then
     delete ea;
     delete e
  endif
  ```
- update primitives: updating the value of sub-level attribute $A.A_i$ for an instance e of E now consists in updating the instance of EA associated to e accordingly.
  ```
  update e(:$A.A_i$ = $exp$)
  where type(e) = $E$
  ⇒
  ```

$\texttt{ea = } EA\texttt{(}R\texttt{:e);}$
$\texttt{update ea(:}A_i\texttt{ = } exp\texttt{);}$

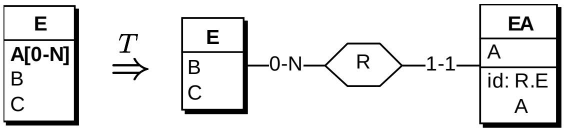
- attribute reference:
  $\texttt{e.}A.A_i \Rightarrow \texttt{e.}R\texttt{.}A_i$
  where type$(\texttt{e}) = E$

Correctness The intuitive justification is exactly the same as for the *ATT-to-ET-inst* transformation. The only difference is that the extracted attribute A is a compound attribute.

10.6.8 Multi-valued attribute to entity type (instance representation)

Signature $\langle R, EA\rangle \leftarrow MultiATT\text{-}to\text{-}ET\text{-}inst(E, A)$

Structural mapping The $MultiATT\text{-}to\text{-}ET\text{-}inst(E, A)$ replaces a multivalued attribute A of an entity type E with another entity type EA, by instance representation. Each instance of entity type E may thus correspond to multiple instances of EA, while each instance of EA is associated to exactly *one* instance of E. As specified in Chapter 2, by default, multivalued attributes represent *sets* of values, i.e. unstructured collections of *distinct* values. Therefore, there cannot exist two distinct instances of EA associated with the same instance of E and having the same value of atomic attribute A. This constraint is expressed through the identifier of EA.



Program transformation The adaptation of the data manipulation primitives involving attribute A is specified below:

- variable declaration :
  $\texttt{a : } E.A\texttt{;} \Rightarrow \texttt{a : } EA.A\texttt{;}$
- record selection expressions: we consider the following abstract selection condition which returns true when the set A contains a given value exp (exp $\in$:A). This condition is now expressed based on the existence of an instance of EA with the proper value of attribute A.
  $E\texttt{(}\ldots exp \in\texttt{:}A\ldots\texttt{)} \Rightarrow E\texttt{(}\ldots R\texttt{:}EA\texttt{(:}A\texttt{ = }exp\texttt{)}\ldots\texttt{)}$
- create primitives: the creation of one instance of E assigning a set of values to attribute A now involves the creation of corresponding instances of EA, linked to the created instance of E through relationship type R.
  $\texttt{create e := } E\texttt{(:}A\texttt{ = }\{exp_{A_1},\ldots,expr_{A_N}\}\texttt{ and :}B\texttt{ = }exp_B\texttt{ and :}C\texttt{ = }exp_C\texttt{)}$

```
⇒
create e := $E$(:$B$ = $exp_B$ and :$C$ = $exp_C$);
create ea(:$A$ = $exp_{A_1}$ and $R$:e);
...
create ea(:$A$ = $exp_{A_N}$ and $R$:e)
```

- delete primitives: deleting an instance of E now implies to remove the associated instances of EA from the database.

```
delete e
where type(e) = E
⇒
for ea := $EA$($R$:e) do
    delete ea
endfor;
delete e
```

- update primitives: We consider three possible update actions on a multivalued attribute A representing a set: (1) assigning a set of values to A ($A = \{v_1, \ldots, v_N\}$), (2) adding a value to the set ($A \cup \{v\}$) and (3) removing a value from the set ($A \setminus \{v\}$). We show below how those update actions translates when expressed on the target schema:
  - assigning a new set of values to attribute A for an instance e of E now necessitates to remove of all instances of EA associated to e, before creating new instances of EA.

    ```
    update e(:$A$ = $\{exp_{A_1}, \ldots, exp_{A_N}\}$)
    where type(e) = E
    ⇒
    for ea := $EA$($R$:e) do
        delete ea
    endfor;
    create ea(:$A$ = $exp_{A_1}$ and $R$:e);
    ...
    create ea(:$A$ = $exp_{A_N}$ and $R$:e)
    ```

  - adding a value to multivalued attribute A for an instance e of E now translates into the creation of a new instance of EA associated to e. This instance is created only if no instance of EA attached e already exists with the same value of A.

    ```
    update e(:$A \cup \{exp\}$)
    where type(e) = E
    ⇒
    if not($EA$(:$A$ = $exp$ and $R$:e)) then
    ```

```
    create ea := $EA$(:$A$ = $exp$ and $R$:e)
endif
```

– removing a value v from multivalued attribute A for an instance e of E can be simulated by deleting the instance of EA associated to e having v as value of attribute A, if any.

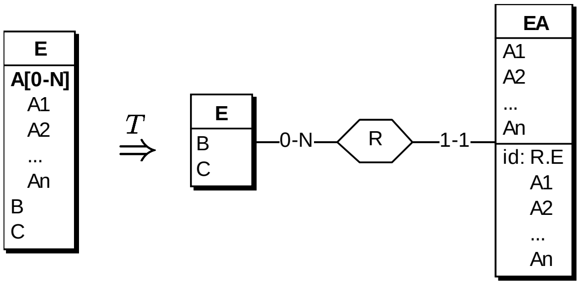
```
update e(:$A \setminus \{exp\}$)
$where\ type(e) = E$
$\Rightarrow$
ea := $EA$(:$A$ = $exp$ and $R$:e);
if (ea $\neq$ null) then
    delete ea
endif
```

Correctness The intuitive justification is exactly the same as for the *ATT-to-ET-inst* transformation, except that the extracted attribute A is a multivalued attribute. As a consequence, simulating the creation of an instance of E potentially necessitates the creation of *several instances* of EA. Conversely, all the instances of EA, that are linked to an instance of E subject to removal, should also be removed.

10.6.9 Compound, multi-valued attribute to entity type (instance representation)

Signature $\langle R, EA\rangle \leftarrow CoMultiATT\text{-}to\text{-}ET\text{-}inst(E, A)$

Structural mapping The $CoMultiATT\text{-}to\text{-}ET\text{-}inst(E, A)$ replaces a multivalued, compound attribute A of an entity type E with another entity type EA, by instance representation. Each instance of entity type E may thus correspond to multiple instances of EA, while each instance of EA is associated to exactly *one* instance of E. As specified in Chapter 2, by default, multivalued attributes represent *sets* of values, i.e. unstructured collections of *distinct* values. Therefore, there cannot exist two distinct instances of EA associated with the same instance of E and having the same combination of values of atomic attributes A_i. This constraint is expressed through the identifier of EA.



Program transformation

- record selection expressions: we consider the following abstract selection condition which returns true when the set A contains a tuple of values for which value of attribute A_i corresponds to value of expression *exp* ($exp \in{:}A.A_i$). This condition is now related to the existence of an instance of EA with the proper value of attribute A_i.
  $E(\ldots exp \in{:}A.A_i \ldots) \Rightarrow E(\ldots R{:}EA({:}A_i = exp)\ldots)$

- create primitives: the creation of one instance of E assigning a set of tuples to attribute A now involves the creation of corresponding instances of EA, linked to the created instance of E through relationship type R.

```
create e := $E$(:$A$ = $\{\langle exp_{A_{1_1}},\ldots,expr_{A_{n_1}}\rangle$
                       $\ldots$
                       $\langle exp_{A_{1_N}},\ldots,expr_{A_{n_N}}\rangle\}$ and
                :$B$ = $exp_B$ and
                :$C$ = $exp_C$)
$\Rightarrow$
create e := $E$(:$B$ = $exp_B$ and :$C$ = $exp_C$);
create ea(:$A_1$ = $exp_{A_{1_1}}$ and...:$A_n$ = $exp_{A_{n_1}}$ and $R$:e);
...
create ea(:$A_1$ = $exp_{A_{1_N}}$ and...:$A_n$ = $exp_{A_{n_N}}$ and $R$:e)
```

- delete primitives: deleting an instance of E now implies to remove the associated instances of EA from the database.

```
delete e
where type(e) = $E$
$\Rightarrow$
for ea := $EA$($R$:e) do
    delete ea
endfor;
delete e
```

- update primitives: We consider three possible update actions on a multivalued compound attribute A representing a set: (1) assigning a set of tuples to A ($A = \{\langle v_{1_1},\ldots,v_{n_1}\rangle,\ldots,\langle v_{1_N},\ldots,v_{n_N}\rangle\}$), (2) adding a tuple to the set ($A \cup \{\langle v_1,\ldots,v_n\rangle\}$) and (3) removing a tuple from the set ($A \setminus \{\langle v_1,\ldots,v_n\rangle\}$). We show below how those update actions translates when expressed on the target schema:
  - assigning a new set of tuples to attribue A for an instance e of E now necessitates to remove of all instances of EA associated to e, before creating new instances of EA.

```
update e(:$A$ = $\{\langle exp_{A_{1_1}}, \ldots, exp_{A_{n_1}}\rangle$
                  ...
                  $\langle exp_{A_{1_N}}, \ldots, exp_{A_{n_N}}\rangle\}$)
$\mathit{where\ type}$(e) $= E$
$\Rightarrow$
for ea := $EA$($R$:e) do
    delete ea
endfor;
create ea(:$A1$ = $exp_{A_{1_1}}$ and ...:$An$ = $exp_{A_{n_1}}$ and $R$:e);
...
create ea(:$A1$ = $exp_{A_{1_N}}$ and ...:$An$ = $exp_{A_{n_N}}$ and $R$:e)
```

- adding a tuple to multivalued, compound attribute A for an instance e of E now translates into the creation of a new instance of EA associated to e. This instance is created only if no instance of EA attached e already exists with the same values of $A1, \ldots, A_n$.

```
update e(:$A \cup \{\langle exp_{A_1}, \ldots, exp_{A_n}\rangle\}$)
$\mathit{where\ type}$(e) $= E$
$\Rightarrow$
if not($EA$(:$A_1$ = $exp_{A_1}$ and ... :$A_n$ = $exp_{A_n}$ and $R$:e)) then
   create ea := $EA$(:$A_1$ = $exp_{A_1}$ and ... :$A_n$ = $exp_{A_n}$ and $R$:e)
endif
```

- removing a tuple of values from multivalued attribute A for an instance e of E can be simulated by deleting the instance of EA associated to e having those values for attributes $A_1, \ldots, A_n$, if any.

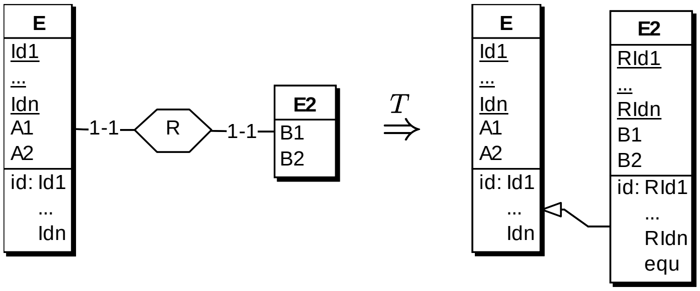
```
update e(:$A \setminus \{\langle exp_{A_1}, \ldots, exp_{A_n}\rangle\}$)
$\mathit{where\ type}$(e) $= E$
$\Rightarrow$
ea := $EA$(:$A_1$ = $exp_{A_1}$ and ... :$A_n$ = $exp_{A_n}$ and $R$:e);
if (ea $\neq$ null) then
   delete ea
endif
```

Correctness The program adaptation rules defined for the *CoMultiATT-to-ET-inst* transformation actually combine the rules propagating the *CoATT-to-ET-inst* and the *MultiATT-to-ET-inst* transformations, but the general principles remain the same.

10.6.10 One-to-one relationship type to foreign key

Signature $\langle\{RId_1, .., RId_n\}, \{Id_1, ..., Id_n\}\rangle \leftarrow \mathit{One2OneRT\text{-}to\text{-}FK}(R, E_2)$

Structural mapping The *One2OneRT-to-FK* transformation replaces a one-to-one relationship type R between entity types E and E_2 with an *identifying* foreign key $\{RId_1, ..., RId_n\}$ in entity type E_2 that references the identifier $\{Id_1, ..., Id_n\}$ of entity type E. The foreign key is subject to an equality constraint with its target identifier. Indeed, the one-to-one relationship type express that each instance of E_2 corresponds to exactly one instance of E and, conversely, each instance of E corresponds to exactly one instance of E_2.



Program transformation

- record selection expressions: The use of relationship type R in a relationship condition must be replaced with a condition based on the equality of the created foreign key and the target identifier. We distinguish two cases: (1) the relationship condition is used for selecting instances of E_2 and (2) it is used for selecting instances of E:

  - $E_2(R\texttt{:e}) \Rightarrow E_2(\texttt{:}RId_1 \texttt{ = e.}Id_1 \texttt{ and } \ldots \texttt{ and :}RId_n \texttt{ = e.}Id_n)$ *where type*$(\texttt{e}) = E$
  - $E(R\texttt{:e2}) \Rightarrow E(\texttt{:}Id_1 \texttt{ = e2.}RId_1 \texttt{ and } \ldots \texttt{ and :}Id_n \texttt{ = e2.}RId_n)$ *where type*$(\texttt{e2}) = E_2$

  The rules above assume relationship conditions of the form R`:var`. More complex conditions of the form R:*recordSelectionExpression* necessitate pre-processing. For instance, the following code fragment:

```
for e2 := $E2(R{:}E(:A_1$ = $exp))$ do
   sequence
endfor
```

  can be rewritten as follows:

```
for e := $E(:A_1$ = $exp)$ do
   e2 := $E_2(R$:e);
   if (e2 $\neq$ null) then
      sequence
   endif
endfor
```

- create primitives: the creation of one instance of E_2 in relationship with an instance of E through relationship type R can be translated as follows.

  ```
  create e2 := $E_2$(:$B_1$ = $exp_{B_1}$ and :$B_2$ = $exp_{B_2}$ and $R$:e)
  $where\ type$(e) = $E$
  $\Rightarrow$
  create e2 := $E_2$(:$B_1$ = $exp_{B_1}$
                and :$B_2$ = $exp_{B_2}$
                and :$RId_1$ = e.$Id_1$
                ...
                and :$RId_n$ = e.$Id_n$)
  ```

- delete primitives: deleting an instance of E obviously implies to remove the associated instance of E_2 from the database. However, no transformation is needed since this delete propagation behaviour was already required in the source schema. Any `delete e` primitive should already by preceded by the following code fragment:

  ```
  e2 := $E_2$($R$:e);
  delete e2
  ```

  According to the rule applying to record selection expressions, this code fragment would be translated as follows:

  ```
  e2 := $E_2$(:$RId_1$ = e.$Id_1$ and ... and :$RId_n$ = e.$Id_n$);
  delete e2
  ```

- update primitives: changing the value of the identifier of an instance of E must be propagated on the foreign key value of the corresponding instance of E_2.

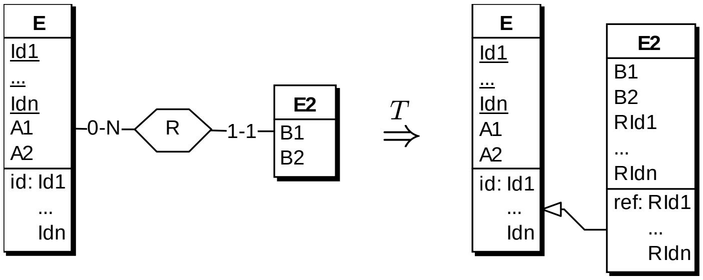
  ```
  update e(:$RId_i$ = $exp$)
  $where\ type$(e) = $E$
  $\Rightarrow$
  e2 := $E_2$(:$RId_1$ = e.$Id_1$ and ... :$RId_n$ = e.$Id_n$);
  update e(:$RId_i$ = $exp$);
  update e2(:$RId_i$ = $exp$)
  ```

Correctness The *One2OneRT-to-FK* transformation propagate to the data level as follows. Each record of type E_2 now has an additional set of attributes, that constitutes a reference to the instance of E it is associated with. As a consequence, each condition based on the relationship between an instance e of E and an instance e_2 of E_2 now translates as the equality between the foreign key value of e_2 and the identifier value of e.

10.6.11 One-to-many relationship type to foreign key

Signature $\langle \{RId_1, .., RId_n\}, \{Id_1, ..., Id_n\} \rangle \leftarrow One2ManyRT\text{-}to\text{-}FK(R, E_2)$

Structural mapping The $One2ManyRT\text{-}to\text{-}FK$ transformation replaces a one-to-many relationship type R between entity types E and E_2 with a foreign key $\{RId_1, .., RId_n\}$ in entity type E_2 that references the identifier $\{Id_1, ..., Id_n\}$ of entity type E.



Program transformation

- record selection expressions: The use of relationship type R in a relationship condition must be replaced with a condition based on the equality of the created foreign key and the target identifier. We distinguish two cases: (1) the relationship condition is used for selecting instances of E_2 and (2) it is used for selecting instances of E:

  – $E_2\texttt{(}R\texttt{:e)} \Rightarrow E_2\texttt{(:}RId_1 \texttt{ = e.}Id_1 \texttt{ and ... and :}RId_n \texttt{ = e.}Id_n\texttt{)}$
  where type(e) $= E$

  – $E\texttt{(}R\texttt{:e2)} \Rightarrow E\texttt{(:}Id_1 \texttt{ = e2.}RId_1 \texttt{ and ... and :}Id_n \texttt{ = e2.}RId_n\texttt{)}$
  where type(e2) $= E_2$

- create primitives: the creation of one instance of E_2 in relationship with an instance of E through relationship type R can be translated as follows.

$$\begin{array}{l}
\texttt{create e2 := } E_2\texttt{(:}B_1 \texttt{ = } exp_{B_1} \texttt{ and :}B_2 \texttt{ = } exp_{B_2} \texttt{ and } R\texttt{:e)} \\
\textit{where type}(\texttt{e}) = E \\
\Rightarrow \\
\texttt{create e2 := } E_2\texttt{(:}B_1 \texttt{ = } exp_{B_1} \\
\qquad\qquad\quad \texttt{and :}B_2 \texttt{ = } exp_{B_2} \\
\qquad\qquad\quad \texttt{and :}RId_1 \texttt{ = e.}Id_1\texttt{)} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad\quad \texttt{and :}RId_n \texttt{ = e.}Id_n\texttt{)}
\end{array}$$

- delete primitives: deleting an instance of E obviously implies to remove the associated instances of E_2 from the database. However, no transformation is needed since this delete propagation behaviour was already required in the source schema. Any `delete e` primitive should already by preceded by the following code fragment:

  ```
  for e2 := $E_2$($R$:e) do
      delete e2
  endfor
  ```

  According to the rule applying to record selection expressions, this code fragment would be translated as follows:

  ```
  for e2 := $E_2$(:$RId_1$ = e.$Id_1$ and ... and :$RId_n$ = e.$Id_n$) do
      delete e2
  endfor
  ```

- update primitives: changing the value of the identifier of an instance of E must be propagated on the foreign key value of the corresponding instances of E_2.

  ```
  update e(:$RId_i$ = $exp$)
  $where\ type$(e) $= E$
  $\Rightarrow$
  for e2 := $E_2$(:$RId_1$ = e.$Id_1$ and ... :$RId_n$ = e.$Id_n$) do
      update e2(:$RId_i$ = $exp$)
  endfor;
  update e(:$RId_i$ = $exp$)
  ```

Correctness The justification is similar to the one given for the *One2OneRT-to-FK* transformation. The difference in this case is that *several* instances of E_2 may reference the same instance of E.

10.7 Applications

The general co-transformational approach proposed in this chapter can be applied in various contexts, among which schema refactoring, database migration and database design. Indeed, each of those database engineering and evolution processes typically involves a *chain* of semantics-preserving schema transformations. The associated co-transformation rules defined above have to be successively composed such that, at the end of the transformation process, the resulting data manipulation code fragment complies with the target database schema.

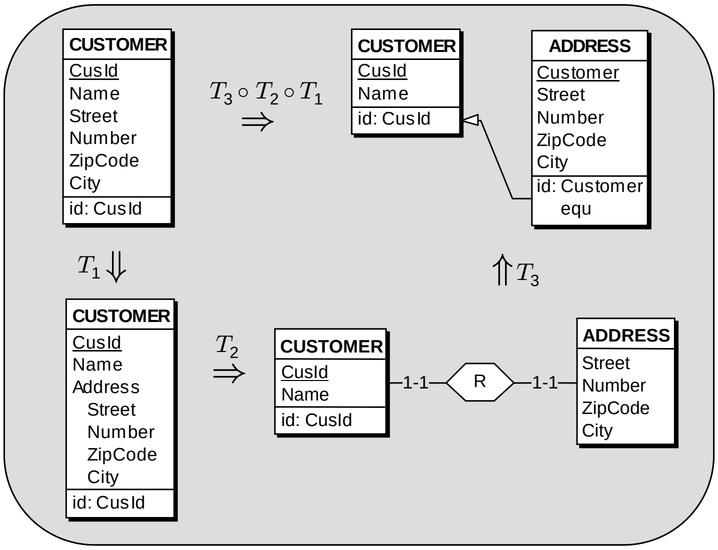


Figure 10.10: Refactoring of a relational logical schema through a chain of GER-to-GER schema transformations.

10.7.1 Application to schema refactoring

Figure 10.10 gives an example of compound schema transformation (or *macro-transformation*), which allows to restructure a relational logical schema. Table `CUSTOMER` is split into two tables. Columns `Street`, `Number`, `ZipCode` and `City` of table `CUSTOMER` are moved to the new table `ADDRESS`. The latter also contains an identifying foreign key column `Customer`, that references the identifier (`CusId`) of the corresponding customer.

This transformation, as depicted in the figure, can be decomposed into three successive primitive transformations:

1. $T_1 \equiv$ `Address` $\leftarrow$ *Aggregate-ATT*(`CUSTOMER`,{`Street`,`Number`,`ZipCode`,`City`}), which consists of the aggregation of columns `Street`, `Number`, `ZipCode` and `City` within an additional compound attribute `Address`.

2. $T_2 \equiv \langle$`R,ADDRESS`$\rangle \leftarrow$ *CoATT-to-ET-inst*(`CUSTOMER`,`Address`), that replaces compound attribute `ADDRESS` with an entity type by instance representation.

3. $T_3 \equiv \langle${`Customer`},{`CusId`}$\rangle \leftarrow$ *One2OneRT-to-FK*(`R`,`ADDRESS`), which converts one-to-one relationship type `R` into foreign key `ADDRESS.Customer`.

Let us now assume that the following `INSERT` statement occurs in the programs:

```
INSERT INTO CUSTOMER (CusId, Name, Street, Number, ZipCode, City)
       VALUES ('C400','Bob','Bob Street','125a','5000','Namur')
```

In LDA, this would be expressed through the following `create` statement:

```
create cus := CUSTOMER(:CusId = 'C400' and
                        :Name = 'Bob' and
                        :Street = 'Bob Street' and
                        :Number = '125a' and
                        :ZipCode = '5000' and
                        :City = 'Namur')
```

The co-transformation rules associated to transformation T_1 states how to adapt the `create` statement accordingly. References to first-level attributes `Street`, `Number`, `ZipCode` and `City` are now expressed as sub-level attribute references.

```
create cus := CUSTOMER(:CusId = 'C400' and
                        :Name = 'Bob' and
                        :Address.Street = 'Bob Street' and
                        :Address.Number = '125a' and
                        :Address.ZipCode = '5000' and
                        :Address.City = 'Namur');
```

Applying the query transformation mapping associated to T_2 leads to the following code fragment, made up of two `create` statements. The second statement creates an instance of entity type `ADDRESS` which now replaces compound attribute `Address` of `CUSTOMER`.

```
create cus := CUSTOMER(:CusId = 'C400' and :Name = 'Bob');
create add := ADDRESS(:Street = 'Bob Street' and
                       :Number = '125a' and
                       :ZipCode = '5000' and
                       :City = 'Namur' and
                      R : cus)
```

Propagating schema transformation T_3 to this code fragment finally produces a SQL-compliant LDA code fragment, where the relationship condition is re-expressed as the equality between the foreign key of the created `ADDRESS` and the identifier of the created `CUSTOMER`:

```
create cus := CUSTOMER(:CusId = 'C400' and :Name = 'Bob');
create add := ADDRESS(:Street = 'Bob Street' and
                       :Number = '125a' and
                       :ZipCode = '5000' and
                       :City = 'Namur' and
                       :Customer = cus.CusId)
```

The SQL translation of the LDA code fragment consists of the two following `INSERT` statements:

```
INSERT INTO CUSTOMER(CusId, Name) VALUES ('C400','Bob');
INSERT INTO ADDRESS(Street, Number, ZipCode, City, Customer)
       VALUES ('Bob Street','125a','5000','Namur', 'C400');
```

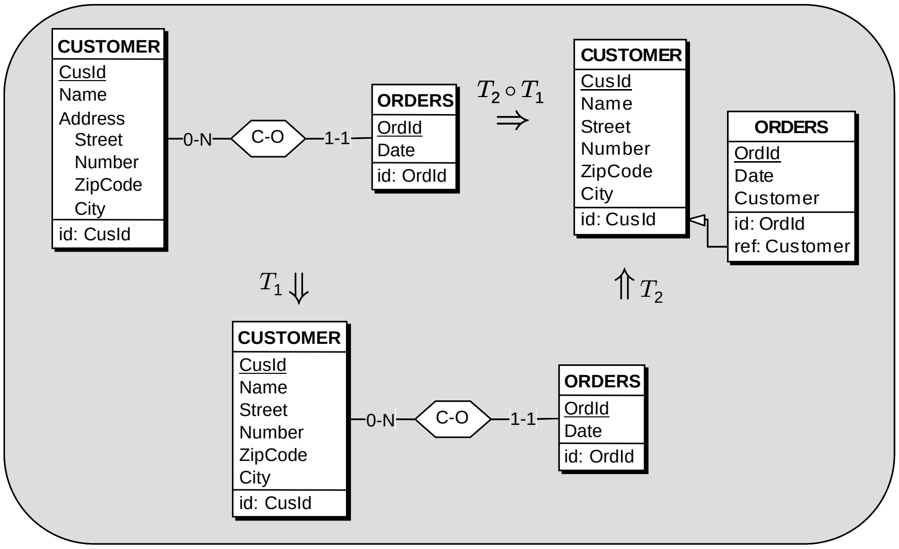


Figure 10.11: Conversion of a CODASYL schema into a relational schema through a chain of GER-to-GER schema transformations.

10.7.2 Application to database migration

Another obvious application of our co-transformational approach is database platform migration. Indeed, this database evolution scenario mainly involves the application of semantics-preserving schema modifications at the logical level. Both the source and target schemas implement the same conceptual schema but in different platform-dependent models. Figure 10.11 considers a simple example of the conversion of a CODASYL schema into an equivalent relational schema. As the Figure shows it, this conversion process consists of two successive GER-to-GER transformations:

1. $T_1 \equiv \{$`Street, Number, ZipCode, City`$\} \leftarrow$ *Disaggregate-ATT*(`CUSTOMER, Address`), that disaggregate compound field `Address`.

2. $T_2 \equiv \langle\{$`Customer`$\}, \{$`CusId`$\}\rangle \leftarrow$ *One2ManyRT-to-FK*(`C-O, ORDERS`), that replaces set type `C-O` with a corresponding foreign key between the member table `ORDERS` and the owner table `CUSTOMER`.

Let us now consider the following `STORE` statement:

```
STORE ORDERS
```

The approximate translation of this statement in LDA is as follows:

```
create ord := ORDERS(:OrdId := uwa-ORDERS-OrdId
                     and :Date := uwa-ORDERS-Date
                     and C-O : curCustomer(C-O))
```

where:

- *uwa*-`ORDERS-OrdId` denotes the current value of field `OrdId` of record `ORDERS` in the user working area (i.e., at the program side);
- *uwa*-`ORDERS-Date` denotes the current value of field `Date` of record `ORDERS` in the user working area (i.e., at the program side);
- *curCustomer*`(C-O)` denotes the current owner of the current set occurence of set type `C-O`.

This statement is not affected by transformation T_1. The co-transformation rules associated to transformation T_2 allow us to adapt the above create statement accordingly. The condition expressed on the one-to-many relationship type `C-O` must be translated into an equality condition between the corresponding foreign key and its target identifier:

```
create ord := ORDERS(:OrdId := uwa-ORDERS-OrdId
               and :Date := uwa-ORDERS-Date
               and :Customer = curCustomer(C-O).CusId)
```

The translation of the resulting create statement into SQL is compatible with the CODASYL-to-SQL translation rule for the `STORE` statement defined in Section 8.4.4:

```
EXEC SQL
    INSERT INTO ORDERS (OrdId, Date, Customer)
    VALUES (:uwa-ORDERS-OrdId, :uwa-ORDERS-Date, curCustomer(C-O).CusId)
END-EXEC
```

10.7.3 Application to database design

In the context of database design, producing a logical schema from a conceptual schema usually involves a chain of semantics-preserving schema transformations. Figure 10.12 depicts an example of such a scenario, where a conceptual schema (the same as in Figure 10.2) expressed in English is translated into a relational logical schema, expressed in French. This logical design process consists of a chain of twenty schema transformations, most of which are renaming transformations:

```
⟨{ORD_NUM},{NUMBER}⟩ ← One2ManyRT-to-FK(ORD-DET, DETAIL)
⟨{PRO_NUM},{NUMBER}⟩ ← One2ManyRT-to-FK(DET-PRO, DETAIL)
⟨{CUS_NUM},{NUMBER}⟩ ← One2ManyRT-to-FK(CUS-ORD, ORDER)
(NUM_PRO) ← RenameATT(PRODUCT,NUMBER)
(NUM_CUS) ← RenameATT(CUSTOMER,NUMBER)
(NUM_ORD) ← RenameATT(ORDER,NUMBER)
(CLIENT) ← RenameET(CUSTOMER)
(COMMANDE) ← RenameET(ORDER)
(PRODUIT) ← RenameET(PRODUCT)
(QUANT_STOCK) ← RenameATT(PRODUIT,STOCK_QTY)
(NUM_CLI) ← RenameATT(CLIENT,NUM_CUS)
```

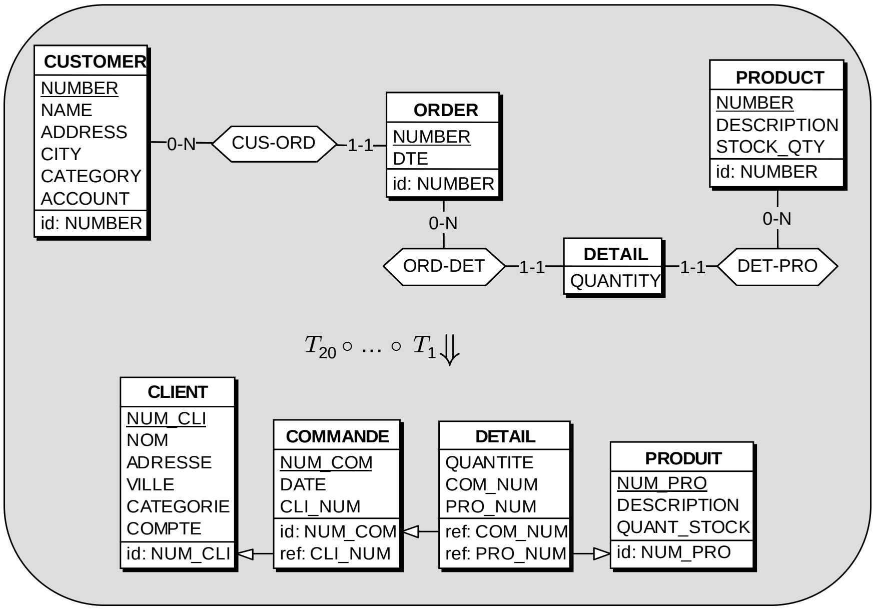


Figure 10.12: Design of a relational logical schema in French from a conceptual schema in English through a chain of GER-to-GER schema transformations.

```
(NOM) ← RenameATT(CLIENT,NAME)
(ADRESSE) ← RenameATT(CLIENT,ADDRESS)
(VILLE) ← RenameATT(CLIENT,CITY)
(CATEGORIE) ← RenameATT(CLIENT,CATEGORY)
(COMPTE) ← RenameATT(CLIENT,ACCOUNT)
(NUM_COM) ← RenameATT(COMMANDE,NUM_ORD)
(CLI_NUM) ← RenameATT(COMMANDE,CUS_NUM)
(QUANTITE) ← RenameATT(DETAIL,QUANTITY)
(COM_NUM) ← RenameATT(DETAIL,ORD_NUM)
```

Figure 10.13 shows the LDA program of Figure 10.4, to which the above schema transformation chain was propagated. This resulting program has the same behaviour as the original program: it still creates a new order (*commande* in French), but it now complies with the relational logical schema of Figure 10.12. In the context of database design, such an transformation-based co-evolution of schema and programs would have interesting benefits. For instance, it would allow programmers to develop *platform-independent* data manipulation algorithms expressed on top of a conceptual schema, those algorithms being then translated into executable programs for a specific database platform. In such an ideal situation, database platform migration would then (simply) require to re-translate the conceptual algorithm by taking into account (1) the new schema transformation chain and (2) the new target query language.

```
// program identification
program NEW_ORDER.

// reference to the underlying GER schema
schema 'create_ord_relational_french.lun';

// declaration of program variables
ord : COMMANDE;
ord-num : COMMANDE.NUM_COM;
ord-date : COMMANDE.DATE;
cus : CLIENT;
cus-num : CLIENT.NUM_CLI;
det : DETAIL;
ord-qty : DETAIL.QUANTITE;
prod : PRODUIT;
prod-code : PRODUIT.NUM_PRO;
is-ok, stop : boolean;
integer : i;

begin
  print('Creating a new order...');
  print('Enter order number:');
  input(ord-num);
  print('Enter order date:');
  input(ord-date);
  is-ok := false;
  // verification loop: the specified customer should exist
  while (is-ok = false) do
      print('Enter customer id:');
      input(cus-num);
      if (CLIENT (:NUM_CLI = cus-num))
          then is-ok := true;
               cus := CLIENT (:NUM_CLI = cus-num)
          else print('Unknown customer !')
      endif
  endwhile;
  // creation of the new order
  create ord := COMMANDE ((:NUM_COM = ord-num)
                   and (:DATE = ord-date)
                   and (:CLI_NUM = cus.NUM_CLI));
  print('Order ', ord.NUM_COM, ' created for customer ', cus.NUM_CLI);
  stop := false;
  // creation of one order detail per ordered product
  while (stop = false) do
      print('Add order detail? (yes=1, no<>1):');
      input(i);
      if (i = 1)
          then  is-ok := false;
                // verification loop: the specified product should exist
                while (is-ok = false) do
                    print('Enter product code:');
                    input(prod-code);
                    if (PRODUIT (:NUM_PRO = prod-code))
                        then  is-ok := true;
                              prod := PRODUIT (:NUM_PRO = prod-code)
                        else  print('Unknown product!')
                    endif
                endwhile;
                print('Enter quantity ordered:');
                input(ord-qty);
                create det := DETAIL((:QUANTITE = ord-qty)
                                 and (:COM_NUM = ord.NUM_COM)
                                 and (:PRO_NUM = prod.NUM_PRO))
          else  stop := true
      endif
  endwhile
end.
```

Figure 10.13: LDA program of Figure 10.4 adapted to the database logical schema of Figure 10.12.

To this end, the extension of our approach would be needed, in order to make it consider (1) the conceptual schema constructs ignored so far (like is-a hierarchies, many-to-many relationship types, and relationship types with attributes) and (2) the GER-to-GER schema transformations that produce and apply to those constructs.

Remark We emphasize the fact that, in practice, we *do not* make use of an intermediate language (like LDA) for supporting the automated adaptation of *real* programs. The only purpose of the examples given in Section 10.7 is to *illustrate* the genericity and the usefullness of the asbtract co-transformation rules provided in this chapter. By instantiating and composing those rules, one can derive *macro-transformation* rules that are suitable for supporting a particular real-life database engineering process (GER-to-relational logical design, relational-to-relational schema refactoring, COBOL-to-relational database migration, CODASYL-to-relational database migration, etc.).

10.8 Tool support

A *proof-of-concept* implementation of the approach presented in this chapter has been developed. As depicted in Figure 10.14, it is based on the ASF+SDF Meta-Environment and DB-MAIN. We defined the syntax of the LDA language using SDF. This allowed us to obtain a working parser for LDA. Then, we specified a set of rewrite rules (in ASF) implementing a limited subset of the co-transformation rules presented in Section 10.6.

The resulting tool takes as input (1) a LDA program complying with a GER schema S_0 and (2) a sequence of n GER transformation signatures expressing the refactoring of schema S_0 into schema S_n. The tool iteratively propagates the schema transformations to the input LDA program, and returns an equivalent output LDA program which conforms to schema S_n.

The automated support also includes a consistency checker, which checks that a LDA program conforms to its underlying GER schema. If this is not the case, it returns a list of structural inconsistencies between the program and the schema, together with the corresponding source-code locations. This consistency checking tool combines the ASF+SDF Meta-Environment (program analysis) and DB-MAIN (schema analysis). In the context of schema evolution, such a tool contributes to analyzing the impact of a schema change on associated programs.

10.9 Related work

The general concept of *coupled software transformation* was defined by Lämmel as follows:

> *"A co-transformation transforms mutually dependent software artifacts of different kinds simultaneously, while the transformation is cen-*

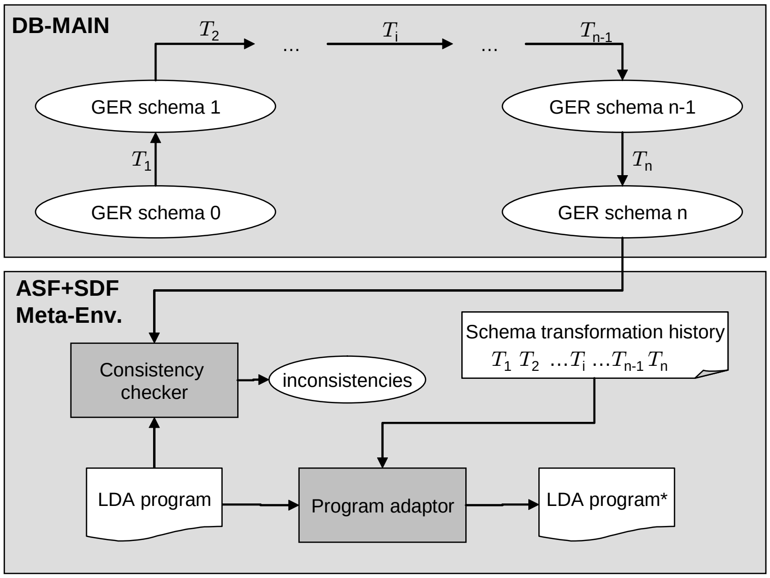


Figure 10.14: Proof-of-concept tool support for database co-transformations.

> *tred around a grammar (or schema, API, or a similar structure) that is shared among the artifacts"*(Lämmel, 2004b).
>
> *"(...) two or more artifacts of potentially different types are involved, while transformation at one end necessitates reconciling transformations at other ends such that global consistency is reestablished"*(Lämmel, 2004a).

The *2LT* project (Cunha et al., 2006; Berdaguer et al., 2007; Alves et al., 2008; Visser, 2008) aims to formalize and to support a particular instance of coupled transformations, namely *two-level transformations*, which involve a transformation on the level of types with transformations on the level of values and operations. The solutions offered by the *2LT* project combine existing techniques of data refinement, typed strategic rewriting, point-free program transformation and advanced functional programming. This generic approach revealed to be applicable to the coupled transformation of database schemas, data instances, queries, and constraints (Visser, 2008).

Lämmel and Lohmann (2001) propose a systematic approach to XML-based *format evolution*. According to this approach (1) format evolution is modelled as step-wise transformations of DTDs and (2) the migration of the corresponding XML documents is largely induced by the DTD-level transformations. The authors identify several categories of transformation steps including renaming, introduction and elimination, folding and unfolding, generalization and restriction, enrichment and removal.

Another application domain of coupled transformations is *grammar evolution*. In this context, Lohmann and Riedewald (2003) tackle the problem of automatically adapting transformation rules after a change in the associated grammar. This general problem typically occurs when trying to adapt an existing program transformation tool to a new version of the programming language of interest. This is particularly true in the case of COBOL program transformation (addressed in this thesis), since there exist a myriad of COBOL dialects.

More recently, Vermolen and Visser (2008) suggest to unify several co-evolution scenarios as format evolution, schema evolution and grammar evolution under the more generic term of *heterogeneous coupled evolution of software languages*. The authors target a systematic approach to the automated support of coupled evolution for any of those scenarios. They propose a generic architecture which takes as input a coupled evolution scenario and a mapping from a top-level transformation to a bottom-level transformation. Based on these two inputs, a structured approach allows (1) the automatic derivation of a transformation language TL for the particular domain, (2) the automatic generation of an interpreter for transformations in TL and (3) the automated propagation of transformation at the top-level to artefacts belonging at the bottom-level.

In our approach, although the modelling language (GER) and the transformation language (GER-to-GER transformations) are fixed, their genericity still allows to cover a large set of schema evolution scenarios. Our co-transformation rules provide an explicit mapping between schema transformations and related program transformations. This mapping relies on *horizontal* consistency preservation rather than on vertical consistency preservation as mainly targeted by Vermolen and Visser (2008).

The *PRISM* system (Curino et al., 2008a) provides a highly integrated support to *relational* schema evolution. This advanced tool suite provides (1) a language for the concise specification of modification operators for relational schemas, (2) impact analysis tools that allow to evaluate the effects of such schema changes, (3) automatic data migration support, (4) optimized translation of old queries to work on the new schema and (5) full documentation of the changes involved by the schema evolution. The schema modifications operators considered are not all semantics-preserving. Query adaptation derives from the schema modification operators and combines (1) a technique called *chase and back-chase* for query rewriting and (2) the generation of SQL views.

With respect to the *PRISM* system, our co-transformational approach considers *finer-grained* and more *generic* schema modifications, that are formally expressed as GER-to-GER transformations. The composition of these basic and generic transformations allow to model several database evolution scenarios among which relational schema refactoring. However, since our rules are based on an abstract data manipulation language, the proof-of-concept implementation of our approach cannot be directly applied to *concrete* database evolution processes, as the *PRISM* system can be.

GER constructs	Relational model	COBOL file model	CODASYL model	Selected?
Entity types	✓	✓	✓	✓
Is-a hierarchies				
Simple attributes	✓	✓	✓	✓
Compound attributes		✓	✓	✓
Multivalued attributes		✓	✓	✓
One-to-one rel. types				✓
One-to-many rel. types			✓	✓
Many-to-many rel.types				
N-any rel. types			(✓)	
Rel. types with attrib.				
Foreign keys	✓			✓

Figure 10.15: The GER constructs necessary to represent relational, COBOL and CODASYL schemas, as compared to the GER constructs we selected.

10.10 Discussion

In this discussion section, we will (1) try to clarify the actual objectives and scope of this chapter, (2) justify the successives choices we have made and (3) better highlight the existing links between this work and the previous chapters.

What are the exact goals of this chapter?

The main objective of this chapter is to address the general problem of adapting programs to database schema refactoring. The term *database schema refactoring* is defined here as the application of *semantics-preserving* transformations to a database schema. Such semantics-preserving transformations:

1. may be involved in various contexts, including database design, database restructuring and database migration;
2. may be applied to different kinds of schemas, including entity-relationship schemas, relational schemas, CODASYL schemas, or COBOL file structures.

Therefore, this chapter aimed to propose a generic approach based on the use of a generic pivot data model: the GER model. As depicted in Figure 10.15, we actually selected a *subset* of the GER constructs, that is sufficient to represent the three main kinds of schemas we have encountered in this thesis, namely relational schemas, COBOL schemas and CODASYL schemas. The main restriction we made was to ignore *multi-member* CODASYL set types (kinds of n-ary relationship types) that are rarely used in practice[8].

[8] Note that it was proposed to remove this complex schema construct from the CODASYL model. This was subject to intense discussions in the late 70's (Bachman, 1977)

Another important requirement was to reach a *fine-level of granularity*, by considering the main *primitive* schema transformations that real-life schema refactoring scenarios generally involve (see Figures 10.10, 10.11 and 10.12 for concrete examples). Our goal was then to associate a set of elementary program transformation rules to those primitive schema transformations. The elementary program adaptation rules obtained may then be composed in order to propagate *chains* of schema transformations to the program level.

In summary, the schema transformations considered in this chapter are primitive, semantics-preserving, GER-to-GER schema transformations that are (directly or indirectly) applicable to the subset of GER schema constructs we selected (and thus to relational, COBOL and CODASYL schemas).

Why an intermediate data manipulation language?

The desired levels of genericity and granularity motivated the use of an *intermediate* data manipulation language for specifying our co-transformation rules. For instance, applying a primitive GER schema transformation to a relational schema may result in a schema that is (temporarily) not relational anymore (see Figure 10.10 for a concrete example). In this case, the corresponding query adaptation rules cannot be expressed on top of the SQL language.

Hence the need for an intermediate data manipulation language that could be used to specify queries against any schemas complying with the GER model subset defined above. This intermediate language should, at least, allow the selection, creation, deletion and modification of database records according to entity-based, attribute-based and relationship-based predicates.

Why LDA?

The LDA language has the merit to meet the above requirement. However, we emphasize the fact that LDA should not be regarded as an attempt to abstract all possible data manipulation languages within a single generic language (this is probably impossible). For instance, this language largely ignores such DML-specific aspects as reading sequence order, error handling or currency indicator management, by focussing on the *structural* aspects shared among most data manipulation languages. In addition, LDA is a navigational language where records are accessed on a *one-record-at-a-time* basis. Nested queries are allowed, but more powerful constructs such as joins are not supported.

Actually, LDA is not *only* a data manipulation language. It also plays the role of *host* programming language, by offering usual language constructs like types, assignments, conditional statements and loops. Some of those additional constructs proved to be useful for expressing our program adaptation rules. For instance, converting a multivalued, compound attribute into a new entity type typically requires the introduction of additional loops in related programs.

In summary, the only purpose of the LDA language is to serve as a conceptual tool on top of which co-transformation rules can be specified in an abstract, yet readable way. Obviously, various other languages and higher-level formalisms could have been used in this work, among which logical languages, functional languages[9], query languages for extended ER schemas (Hohenstein and Engels, 1992; Lawley and Topor, 1994), first-order logic, OCL and description logics. Further investigations are needed to assess and compare the suitability of those alternative formalisms for our purpose. We do not claim that LDA is the best choice we could make, but we believe it constitutes a good tradeoff between mathematical formalisms and executable languages.

Such a tradeoff is required, since the ultimate objective of our co-transformation rules is to serve as generic reference when building automated program adaptation support for plaform-specific schema refactoring scenarios. At the time of writing this thesis, this objective seemed to be (partially) reached. Indeed, our co-transformational approach was recently recognized by the developers of the *PRISM* system, as one of "*the most relevant approaches to the general problem of schema evolution*" providing "*solid theoretical foundations and interesting methodological approaches*" (Curino et al., 2008a). In addition, as already mentionned, we also implicitly made use of our co-transformation rules during the development of COBOL-to-relational and CODASYL-to-relational wrapper generators.

How does this chapter relate to previous chapters?

Depending on the point of view adopted, our co-transformational approach to schema refactoring can be seen either as a generalisation or as a specialisation of our results on database platform migration (Chapters 4, 7 and 8). On the one hand, it is more generic, since it does not make any assumption about the source and target data models. Indeed, the GER model allows to cover most of the existing database paradigms, while the considered schema transformations are reusable in various database evolution scenarios, including database migration. On the other hand, it is more specific, since the general scenario studied does not involve the replacement of the data manipulation language. The program adaptation rules are, indeed, defined on top of the LDA language.

10.11 Conclusions

This chapter has presented a general co-transformational approach to database schema refactoring. According to this approach a semantics-preserving schema transformation is defined as the application of coupled transformations, that modify the database schema, the data instances and the related programs so that the global consistency is preserved. A set of representative co-transformation rules and a prototype tool have been presented. The application of the approach was illustrated in three particular scenarios, namely schema refactoring, database migration and database design.

[9] as in the *2LT* project (Visser, 2008).

Part VI

Conclusions

Chapter 11

Conclusions

I love it when a plan comes together
– Colonel John "Hannibal" Smith

This chapter concludes the thesis. It summarizes the research contributions made while trying to answer our research questions and elaborated on the main lessons we learned from this work. Avenues for future research in the domain of data-intensive systems evolution are also discussed.

11.1 Summary of the contributions

Research question 1

Can automated program analysis techniques help to recover implicit knowledge on the structure and constraints of a database?

Chapter 5 presented a tool-supported dataflow analysis approach in the context of database reverse engineering. This approach consists in statically analyzing *intra-query* data dependencies, i.e, dependencies that are involved in the execution of database queries. Through the generalization of the term *database query*, the approach has been extended in order to face the frequent situation where one or several data access module(s) are used to access the database. Industrial reverse engineering projects has shown that the suggested approach, and its supporting tools, may allow the recovery of implicit knowledge on the database structures and constraints (undeclared foreign keys, finer-grained decomposition and more expressive names for record types and fields).

In Chapter 6, we provided an in-depth exploration of the use of dynamic program analysis techniques for reverse engineering relational databases. Those techniques particularly target the analysis of data-intensive systems in the presence of *automatically generated* SQL queries. First, we identified, illustrated and compared a set of techniques for *capturing* the SQL queries executed at runtime. Then, we elaborated on the analysis of SQL traces in the context of database reverse engineering. We particularly focused on implicit foreign key detection, by identifying related heuristics for trace analysis, which combine both intra-query dependencies (SQL joins) and inter-query dependencies (input-input and output-input dependencies). An initial experiment, based on a real-life application, allowed us to establish the analysis of SQL execution traces as a very promising technique for relational database reverse engineering.

Both chapters showed that automated program analysis techniques may significantly contribute to the enrichment of the database schema, especially through the identification of intra-query and inter-query dependencies. Analyzing those dependencies allow, in a second stage, to reveal implicit links (1) between schema constructs and (2) between schema constructs and program variables. We also clearly observed the interesting complementarity of static program analysis and dynamic program analysis for database reverse engineering. Static analysis has the merit of considering the *complete* set of source code files, but its scope is limited to what is statically decidable. Dynamic analysis naturally yields to *partial* results, since it is limited to particular execution scenarios, but the underlying techniques may allow to capture information that is out of the scope of static analysis.

Research question 2

What are the possible strategies for migrating a legacy data-intensive system towards a modern database platform? How do they compare?

Chapter 4 addressed this question, by developping a comprehensive framework for the migration of legacy data-intensive systems. This framework identifies six representative migration strategies, relying on two dimensions: database dimension and program dimension. Based on a common running example, the two database conversion strategies and the three program conversion strategies identified have been illustrated and compared. We learned, in particular, that database conversion may greatly benefit from an initial database reverse engineering process, as opposed to a one-to-one schema conversion strategy. As far as program conversion is concerned, the more seducing strategy (*Logic rewriting* or P3), that consists in adapting the logic of the programs to the target database structure and language, is also the only one that cannot be (easily) automated. By contrast, the *Wrapper strategy* (P1) permits a high level of automation while minimizing the adaptation of the legacy programs.

Research question 3

Is it possible to automatically adapt large legacy systems to the migration of their underlying database?

The only way to prove that it is possible to automatically adapt a system to the migration of its database it to develop methods and tools allowing to do so, and to apply them to convincing case studies. This thesis considered two representative migration scenarios, COBOL-to-relational migration (Chapter 7) and CODASYL-to-relational migration (Chapter 8). For each scenario, (1) we illustrated to main challenges involved by the scenario, (2) we provided systematic translation rules for simulating the legacy data manipulation primitives on top of a relational database, and (3) we described a set of tools supporting the automated migration of the legacy programs. As far as validation is concerned, Chapter 9 reported on two industrial migration projects for which the approach and tools presented of Chapter 8 have been used successfully. Both projects aimed at migrating a CODASYL (IDS/II) database towards a relational (DB2) database platform. Although we were able to adapt the legacy programs with a high level of automation, the replacement of some CODASYL primitives required manual intervention.

Research question 4

How to preserve the consistency between an evolving database schema and associated queries?

In Chapter 10, we presented a tool-supported co-transformational approach to the propagation of database schema changes on application programs. This approach consists in systematically associating abstract program transformation rules to a set of semantics-preserving schema transformations. A generic model (GER) was chosen as an pivot model for database schemas and an abstract language (LDA) was used for specifying the program transformation rules. We showed the this combination allows our abstract rules to be instantiated in the context of several database engineering and evolution processes, including schema refactoring, database migration and database design. From this work, we learned that coupled transformations form a sound basis for supporting the co-evolution of databases and related programs.

Links between research questions

More generally, this thesis has shown that automated program analysis and transformation techniques may significantly contribute to the general process of database evolution. The program analysis techniques support the database reverse engineering phase, by recovering implicit knowledge about the database subject to evolution. The program transformation techniques support the program adaptation step, by propagating the evolution of the database to the program level.

Several important links exist between the contributions made with respect to the above research questions (RQ):

- RQ1 vs RQ2 : the automated program analysis techniques developed in Chapters 5 and 6 aim to support the database reverse engineering process, which in turn constitutes the main basis of the conceptual database conversion strategy identified in Chapter 4.

- RQ2 vs RQ3 : the automated program adaptation techniques presented in Chapters 7 and 8 actually implement two program conversion strategies described in Chapter 4 in the particular context of COBOL-to-relational and CODASYL-to-relational migration, respectively.

- RQ3 vs RQ4 : the database migration process typically involves the refactoring of the database schema, which in turn necessitates the adaptation of the queries occurring in the programs. The generic co-transformation rules specified in Chapter 10, once composed and instantiated, form a sound basis for automating such a program adaptation.

11.2 Lessons learned

In this section, we elaborate on the main lessons we learned from this research work.

Automation is necessary but not sufficient The industrial case studies presented in Chapters 5 and 9 confirmed that data-intensive system evolution requires scalable and reusable tool support. But they also showed that the full-automation of the process is clearly unreachable, since human intervention may be required at almost every step. Database reverse engineering is generally an iterative process, where analysts formulate and validate hypotheses based on the automated analysis of programs and database contents. The database conversion phase may also involve multiple human decisions, taking several aspects into account like the existence of naming and structural conventions, the presence of data inconsistencies or the need for high performance. The program adaptation step may also necessitate manual intervention, depending on the data manipulation language and the type of queries used by the application programs.

Mappings are everywhere The concept of *mapping* is omnipresent in the domain of data-intensive system evolution. In our work, we have encountered multiple kinds of mappings, among which inter-model mappings, inter-schema mappings, intra-schema mappings, inter-query mappings, intra-query mappings, inter-language mappings and inter-paradigmatic mappings. In our research domain, several important challenges are actually related to the *definition*, *detection*, *evolution*, *exploitation*, and *visualisation* of such mappings.

Transformation and generation are good colleagues Another important observation we made concerns the advantage of combining generative and transformational techniques when migrating programs towards a new database platform. Indeed, Chapter 9 showed that such a combination allows to make the migration process more flexible. According to this approach, the legacy programs are transformed only once, independently on the database migration process. In contrast, the generated code can be re-generated as many times as necessary, without additional alteration of the legacy code.

To wrap, or not to wrap: that is the question In the context of database migration, the use of wrapping techniques has the merit of minimizing the modification of the legacy application programs. The major complexity of the process is encapsulated within the wrappers, that deal with both structural conversion and language translation issues. However, the wrapper-based program conversion strategy also suffers from several drawbacks, among which performance degradation. In our opinion, this strategy should be regarded as a short-term solution allowing the application programs to be rapidly interfaced with the target database at low cost and at low risk. As a second stage, the legacy programs may then be incrementally rewritten so that they can access the migrated database in a more direct, natural and efficient way.

Migration is an opportunity to improve Migrating a software system towards a new database platform aims to obtain the same software system, that offers the same functionalities and manipulates the same data, except that the data are now stored in a new database. Besides its main objective, migration may also constitute an excellent opportunity to improve the quality of the system. For instance, discovering an implicit foreign key allows, as a second step, to make quality improvements at different levels: (1) at the schema level, by making the referential constraint explicit; (2) at the data level, by detecting and correcting the data instances that violate the referential constraint; and (3) at the program level, by improving the management of the referential constraint wherever necessary.

Industry is a great laboratory A large part of our research was carried out in cooperation with ReVeR, our industrial partner. In this context, we mainly adopted the *Industry-as-Laboratory* research approach proposed by Potts (1993). This approach is based on the close involvement of the researcher with industrial projects, which allows him (1) to identify research problems of interest, (2) to elicit realistic requirements for the solutions and (3) to validate the proposed solutions. In other words, this research style considers industrial case studies as a vehicule for conducting research, not merely as a way to demonstrate the value of research results. We believe that this cooperation mode (1) is beneficial for both industrial and academic partners and (2) is particularly well-suited for research in software maintenance and evolution, as already experienced by other PhD researchers in the past (Veerman, 2007; Bruntink, 2008b).

11.3 Open issues and future challenges

The future work will consolidate and extend the promising results obtained so far. We anticipate below several research directions together with some of their challenges.

Supporting other evolution scenarios In this thesis, we mainly focused on database evolution scenarios that perserve the semantics of the database schema. The techniques and tools we have presented allow the automatic adaptation of application programs to (1) *schema refactoring*, (2) *platform migration* or (3) a combination of them. Supporting program adaptation in the case of non-semantics-preserving schema changes appears as another interesting challenge. This would necessitate more sophisticated, yet less automatable, program analysis and transformation techniques.

Supporting data quality refactorings The thesis addresses the propagation of *structural* schema refactorings. One could also consider another kind of schema refactorings, namely *data quality refactorings* (Ambler and Sadalage, 2006), which aim at improving the overall quality of the database contents. Examples of such refactorings for relational databases are the introduction of a common format for a given column, the replacement of a nullable column with a non-nullable column, and the introduction of a column constraint. The adaptation of the programs mainly consists in adding a proper exception handling code, at every program point where the corresponding table is modified.

Supporting other processes In the context of the co-evolution of databases and programs, another important process is *impact analysis*. When changes are to be applied to a database schema, an in-depth analysis of the impact of such changes on the programs allows to better evaluate the program adaptation effort. Several mature tool-supported approaches have been proposed for specific platforms (Karahasanovic, 2002; Maule et al., 2008; Papastefanatos et al., 2008). We believe that these approaches could be generalized by explicitly expressing, in a generic way, the consistency relationships that hold between the query language and the data model of interest. We are currently working on this topic.

Supporting other program conversion strategies In this thesis, our approach to the automated adaptation of programs consisted in *translating* queries accessing the source database into equivalent queries on the target database, while *preserving* the data manipulation logic of the programs. One could also further investigate other program conversion strategies, according to which the data manipulation logic of programs is *adapted* to the target database paradigm (like the *Logic Rewriting* strategy of Chapter 4). Our industrial experience taught us that such strategies are not easily automatable. Although some partial techniques have been proposed in this direction (Katz and Wong, 1982), they usually assume the application programs to conform to a *canonical* form. Unfortunately, such an assumption is not realistic since the level of *variability* of data processing code is typically very high. This clearly constitutes the major obstacle to *logic-based* program adaptation. Similar variability problems were encountered by Bruntink et al. (2007) in the context of the renovation of cross-cutting concern code (Bruntink, 2008b).

Supporting program analysis and transformation for other platforms A large part of the thesis was dedicated to the analysis and transformation of *legacy COBOL systems*. It is obvious that data-intensive systems developed on top of more modern platforms also require similar support for the co-analysis and the co-evolution of database and programs. For instance, web-based applications typically rely on poorly documented databases, as seen in Chapter 6. Other co-evolution challenges will have to be faced with the introduction of O/R mapping technologies like Hibernate, where database accesses are implicitly generated. While they seem very convenient for the programmer, studying how such technologies affect the effort needed when evolving the database still remains to be done.

Analyzing the co-evolution of databases and programs This thesis aimed to *support* the co-evolution of databases and programs. Another research direction would be to *observe* how databases and programs co-evolve over time in practice (Curino et al., 2008b; Lin and Neamtiu, 2009). A possible approach consists in mining real-life software repositories. Analyzing the data that is stored in version control systems (like CVS or SVN) could allow to teach us several important lessons on (1) the decreasing/increasing quality/complexity of program-database mappings, (2) the most frequently applied database evolutions, (3) the way database changes are propagated to programs.

Extending the scope of dynamic analysis for database queries As discussed in Chapter 6, dynamic analysis of SQL queries have a wide range of applications. We intend to consolidate and extend our initial results, by exploring the use of dynamic analysis in other domains than database reverse engineering, including quality assessment for database queries, data-intensive program comprehension, data security and consistency management. Furthermore, we would like to investigate the benefits of combined approaches, based on the co-analysis of schemas, data, programs and query execution traces.

Bibliography

Agrawal, H., 1994. On slicing programs with jump statements. In: PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation. ACM Press, New York, NY, USA, pp. 302–312.

Agrawal, H., Horgan, J. R., 1990. Dynamic program slicing. In: PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation. ACM, New York, NY, USA, pp. 246–256.

Alves, T. L., Silva, P. F., Visser, J., 2008. Constraint-aware schema transformation. Electr. Notes Theor. Comput. Sci.To appear.

Ambler, S. W., Sadalage, P. J., 2006. Refactoring Databases: Evolutionary Database Design. Addison-Wesley.

Andersson, M., 1998. Searching for semantics in cobol legacy applications. In: Spaccapietra, S., Maryanski, F. J. (Eds.), Data Mining and Reverse Engineering: Searching for Semantics, IFIP TC2/WG2.6 Seventh Conference on Database Semantics (DS-7). Vol. 124 of IFIP Conference Proceedings. Chapman & Hall, pp. 162–183.

Bachman, C. W., 1977. Why restrict the modelling capability of codasyl data structure sets? In: Proceedings of the AFIPS National Computer Conference. Vol. 46 of AFIPS Conference Proceedings. pp. 69–75.

Ball, T., Horwitz, S., Dec. 1992. Slicing programs with arbitrary control flow. Tech. Rep. TR1128, University of Wisconsin.

Balzer, R., 1991. Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering (ICSE'91-. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 158–165.

Batini, C., Ceri, S., Navathe, S. B., 1992. Conceptual Database Design : An Entity-Relationship Approach. Benjamin/Cummings.

Baxter, I. D., Pidgeon, C., Mehlich, M., 2004. DMS: Program transformations for practical scalable software evolution. In: Proceedings of the 26th International Conference on Software Engineering (ICSE'04). IEEE Computer Society, pp. 625–634.

Beck, J., Eichmann, D., 1993. Program and interface slicing for reverse engineering. In: ICSE '93: Proceedings of the 15th international conference on Software Engineering. IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 509–518.

Behm, A., Geppert, A., Dittrich, K., December 1997. On the migration of relational schemas and data to object-oriented database systems. In: Proceedings of the 5th International Conference on Re-Technologies in Information Systems. Klagenfurt, Austria, pp. 13–33.

Berdaguer, P., Cunha, A., Pacheco, H., Visser, J., 2007. Coupled schema transformation and data conversion for XML and SQL. In: Hanus, M. (Ed.), Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages. Vol. 4354 of LNCS. Springer-Verlag, pp. 290–304.

Bianchi, A., Caivano, D., Visaggio, G., 2000. Method and process for iterative reengineering of data in a legacy system. In: Proc. Working Conf. Reverse Engineering (WCRE). pp. 86–97.

Binkley, D., Ceccato, M., Harman, M., Ricca, F., Tonella, P., 2006. Tool-supported refactoring of existing object-oriented code into aspects. IEEE Transactions on Software Engineering 32 (9), 698–717.

Bisbal, J., Lawless, D., Wu, B., Grimson, J., September/October 1999. Legacy information systems: Issues and directions. IEEE Software 16 (5), 103–111.

Blaha, M. R., Premerlani, W. J., 1995. Observed idiosyncracies of relational database designs. In: Proceedings of the Second Working Conference on Reverse Engineering (WCRE'95). IEEE Computer Society, Washington, DC, USA, p. 116.

Bravenboer, M., Kalleberg, K. T., Vermaas, R., Visser, E., 2008. Stratego/xt 0.17. a language and toolset for program transformation. Sci. Comput. Program. 72 (1-2), 52–70.

Brodie, M. L., Stonebraker, M., 1995. Migrating Legacy Systems. Gateways, Interfaces, and the Incremental Approach. Morgan Kaufmann Publishers.

Brown, G. D., 1998. Advanced Cobol For Structured and Object Oriented Programming, 3rd Edition. John Wiley.

Bruntink, M., 2008a. Reengineering idiomatic exception handling in legacy c code. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08). IEEE Computer Society, pp. 133–142.

Bruntink, M., March 2008b. Renovation of idiomatic crosscutting concerns in embedded systems. Ph.D. thesis, Faculty of Electrical Engineering, Mathematics and Computer Science.

Bruntink, M., van Deursen, A., D'Hondt, M., Tourwé, T., 2007. Simple crosscutting concerns are not so simple: analysing variability in large-scale idioms-based implementations. In: Barry, B. M., de Moor, O. (Eds.), AOSD. Vol. 208 of ACM International Conference Proceeding Series. ACM, pp. 199–211.

Bruntink, M., van Engelen, R., Tourwe, T., 2005. On the use of clone detection for identifying crosscutting concern code. IEEE Computer Society Trans. Software Engineering 31 (10), 804–818.

BULL, 2001. DBSP. `http://www.bull.com/servers/gcos8/products/dbsp/dbsp`.

Canfora, G., Cimitile, A., Munro, M., 1996. An improved algorithm for identifying objects in code. Software—Practice and Experience, Wiley 26 (1), 25–48.

Canfora, G., Santo, G. D., Zimeo, E., 2006. Developing and executing Java AWT applications on limited devices with TCPTE. In: Proc. Int'l Conf. Software Engineering (ICSE). ACM Press, New York, NY, USA, pp. 787–790.

Ceccato, M., Marin, M., Mens, K., Moonen, L., Tonella, P., Tourwé, T., 2006. Applying and combining three different aspect mining techniques. Software Quality Control 14 (3), 209–231.

Chikofsky, E. J., 1996. "The Necessity of Data Reverse Engineering". Foreword for Peter Aiken's Data Reverse Engineering. McGraw Hill, Ch. 0, pp. 8–11.

Chikofsky, E. J., Cross, J. H., 1990. Reverse engineering and design recovery: A taxonomy. IEEE Software 7 (1), 13–17.

Clarinval, A., 1981. Comprendre, Connatre et Matriser le Cobol, 2nd Edition. Presses Universitaires de Namur.

Cleve, A., Hainaut, J.-L., 2006. Co-transformations in database applications evolution. In: Lämmel, R., Saraiva, J., Visser, J. (Eds.), Post-proceedings of GTTSE 2005, Generative and Transformation Techniques in Software Engineering, 4–8 July, 2005, Braga, Portugal. Vol. 4143 of Lecture Notes in Computer Science. Springer, pp. 409–421, summer school participant contribution.

Cleve, A., Hainaut, J.-L., 2008. Dynamic analysis of sql statements for data-intensive applications reverse engineering. In: Proceedings of the 15th Working Conference on Reverse Engineering. IEEE, pp. 192–196.

Cleve, A., Henrard, J., Hainaut, J.-L., 2006. Data reverse engineering using system dependency graphs. In: Proceedings of the 13th Working Conference on Reverse Engineering (WCRE'06). IEEE Computer Society, Washington, DC, USA, pp. 157–166.

Cleve, A., Henrard, J., Roland, D., Hainaut, J.-L., 2008a. Wrapper-based system evolution - application to codasyl to relational migration. In: Proceedings of the 12th European Conference in Software Maintenance and Reengineering (CSMR'08). IEEE Computer Society, pp. 13–22.

Cleve, A., Lemaitre, J., Hainaut, J.-L., Mouchet, C., Henrard, J., 2008b. The role of implicit schema constructs in data quality. In: Missier, P., Lin, X., de Keijzer, A., van Keulen, M. (Eds.), Proceedings of the 6th International Workshop on Quality in Databases (QDB'08). pp. 33–40.

Cordy, J. R., Dean, T. R., Malton, A. J., Schneider, K. A., 2002. Source transformation in software engineering using the txl transformation system. Information & Software Technology 44 (13), 827–837.

Cunha, A., Oliveira, J. N., Visser, J., 2006. Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (Eds.), Proceedings of the 14th International Symposium on Formal Methods. Vol. 4085 of LNCS. Springer-Verlag, pp. 284–299.

Curino, C., Moon, H. J., Zaniolo, C., 2008a. Graceful database schema evolution: the prism workbench. Proceedings of the VLDB Endowment 1 (1), 761–772.

Curino, C. A., Moon, H. J., Tanca, L., Zaniolo, C., 2008b. Schema evolution in wikipedia: toward a web information system benchmark. In: Cordeiro, J., Filipe, J. (Eds.), International Conference on Enterprise Information Systems (ICEIS). pp. 323–332.

DB-MAIN, 2006. The DB-MAIN official website. http://www.db-main.be.

de Lucia, A., Lucca, G. A. D., Fasolino, A. R., Guerra, P., Petruzzelli, S., 1997. Migrating legacy systems towards object-oriented platforms. In: Proc. Int'l Conf. Software Maintenance (ICSM). IEEE Computer Society, Los Alamitos, CA, USA, p. 122.

El-Ramly, M., Eltayeb, R., Alla, H., 2006. An experiment in automatic conversion of legacy Java programs to C#. In: Proceedings of IEEE International Conference on Computer Systems and Applications. pp. 1037–1045.

Elmasri, R., Navathe, S. B., 1999. Fundamentals of Database Systems, 3rd Edition. Benjamin/Cummings, Ch. Appendix C: An Overview of the Network Data Model, pp. 917–940.

Embury, S. M., Shao, J., 2001. Assisting the comprehension of legacy transactions. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE'01). IEEE Computer Society, Washington, DC, USA, p. 345.

Englebert, V., 2002. Voyager 2 reference manual. Tech. rep., University of Namur, `http://www.info.fundp.ac.be/~dbm/publication/2002/VOYAGER-2-reference-manual.pdf`.

Gallagher, K. B., Lyle, J. R., Aug. 1991. Using program slicing in software maintenance. IEEE Transactions on Software Engineering 17 (8), 751–761.

Girard, J.-F., Koschke, R., Schied, G., 1999. A metric-based approach to detect abstract data types and state encapsulations. Journal on Automated Software Engineering 6 (4), 357–386.

Hainaut, J.-L., 1986. Conception assistée des applications informatiques - 2: Conception de la base de données. Masson, Paris.

Hainaut, J.-L., 1989. A generic entity-relationship model. In: Proceedings of the IFIP WG 8.1 Conference on Information System Concepts: an in-depth analysis. North-Holland, pp. 109–138.

Hainaut, J.-L., 1996. Specification preservation in schema transformations - application to semantics and statistics. Data Knowledge Engineering 19 (2), 99–134.

Hainaut, J.-L., 2002. Introduction to database reverse engineering. LIBD Publish., `http://www.info.fundp.ac.be/~dbm/publication/2002/DBRE-2002.pdf`.

Hainaut, J.-L., 2006. The transformational approach to database engineering. In: Lämmel, R., Saraiva, J., Visser, J. (Eds.), Generative and Transformational Techniques in Software Engineering. Vol. 4143 of Lecture Notes in Computer Science. Springer-Verlag, pp. 95–143.

Hainaut, J.-L., 2009. Network data model. In: Özsu, M. T., Liu, L. (Eds.), Encyclopedia of Database Systems. Springer, to appear.

Hainaut, J.-L., Chandelon, M., Tonneau, C., Joris, M., May 1993. Contribution to a theory of database reverse engineering. In: Proceedings of the IEEE Working Conf. on Reverse Engineering. IEEE Computer Society Press, Baltimore, pp. 161–170.

Hainaut, J.-L., Cleve, A., Henrard, J., Hick, J.-M., 2008. Migration of legacy information systems. In: Mens, T., Demeyer, S. (Eds.), Software Evolution. Springer, pp. 105–138.

Hainaut, J.-L., Englebert, V., Henrard, J., Hick, J.-M., Roland, D., 1996a. Database reverse engineering: From requirements to care tools. Automated Software Engineering 3, 9–45.

Hainaut, J.-L., Henrard, J., Englebert, V., Roland, D., Hick, J.-M., 2009. Database Reverse Engineering. Springer, pp. 263–263, to appear.

Hainaut, J.-L., Henrard, J., Hick, J.-M., Roland, D., Englebert, V., 1996b. Database design recovery. In: Proceedings of International Conference on Advances Information System Engineering (CAiSE). Vol. 1080 of Lecture Notes in Computer Science. Springer-Verlag, pp. 272–300.

Hainaut, J.-L., Henrard, J., Hick, J.-M., Roland, D., Englebert, V., 2000. The nature of data reverse engineering. In: Proceedings of Data Reverse Engineering Workshop (DRE'2000).

Hainaut, J.-L., Hick, J.-M., Henrard, J., Roland, D., Englebert, V., 1997. Knowledge transfer in database reverse engineering: A supporting case study. In: Proc. Working Conf. Reverse Engineering (WCRE). pp. 194–203.

Halfond, W. G. J., Orso, A., 2005. Combining static analysis and runtime monitoring to counter sql-injection attacks. In: WODA '05: Proceedings of the third international workshop on Dynamic analysis. ACM, New York, NY, USA, pp. 1–7.

Harman, M., Hierons, R. M., 2001. An overview of program slicing. Software Focus 2 (3), 85–92.

Heckel, R., Correia, R., Matos, C., El-Ramly, M., Koutsoukos, G., Andrade, L., 2008. Architectural transformations: From legacy to three-tier and services. In: Mens, T., Demeyer, S. (Eds.), Software Evolution. Springer, pp. 139–170.

Henrard, J., 2003. Program understanding in database reverse engineering. Ph.D. thesis, University of Namur.

Henrard, J., Englebert, V., Hick, J.-M., Roland, D., Hainaut, J.-L., 1998. Program understanding in databases reverse engineering. In: Quirchmayr, G., Schweighofer, E., Bench-Capon, T. (Eds.), Proceedings of 9th International Conference on Database and Expert Systems Applications (DEXA'98). Vol. 1460 of Lecture Notes in Computer Science. Springer, pp. 70–79.

Henrard, J., Roland, D., Cleve, A., Hainaut, J.-L., 2007. An industrial experience report on legacy data-intensive system migration. In: Canfora, G., Tahvildari, L. (Eds.), Proceedings of the 23rd International Conference on Software Maintenance (ICSM'07). IEEE Computer Society.

Henrard, J., Roland, D., Cleve, A., Hainaut, J.-L., 2008. Industrial experiences in data reengineering. In: Proceedings of the 15th Working Conference on Reverse Engineering (WCRE'08). IEEE Computer Society, to appear.

Hick, J.-M., 2001. Evolution d'applications de bases de données relationnelles - méthodes et outils. Ph.D. thesis, University of Namur.

Hick, J.-M., Hainaut, J.-L., December 2006. Database application evolution: A transformational approach. Data & Knowledge Engineering 59, 534–558.

Hohenstein, U., Engels, G., 1992. Sql/eer—syntax and semantics of an entity-relationship-based query language. Inf. Syst. 17 (3), 209–242.

Horwitz, S., Reps, T., Binkley, D., Jan. 1990. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems 12 (1), 26–60.

Jackson, D., Rollins, E. J., 1994. A new model of program dependences for reverse engineering. In: SIGSOFT '94: Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering. ACM Press, New York, NY, USA, pp. 2–10.

Jahnke, J.-H., Wadsack, J. P., May 1999. Varlet: Human-centered tool support for database reengineering. In: Proceedings of Workshop on Software-Reengineering (WCRE'99).

Jeusfeld, M. A., Johnen, U. A., December 1994. An executable meta model for reengineering of database schemas. In: Proceedings of Conference on the Entity-Relationship Approach. Manchester, pp. 533–547.

Johnson, L., 1986. File Techniques for Data Base Organization in Cobol, 2nd Edition. Prentice-Hall International.

Karahasanovic, A., 2002. Supporting application consistency in evolving object-oriented systems by impact analysis and visualisation. Ph.D. thesis, University of Oslo.

Katz, R. H., Wong, E., 1982. Decompiling codasyl dml into relational queries. ACM Trans. Database Syst. 7 (1), 1–23.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., 2001. An overview of aspectj. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP'01). Vol. 2072 of LNCS. pp. 327–353.

Kontogiannis, K., Martin, J., Wong, K., Gregory, R., Müller, H., Mylopoulos, J., 1998. Code migration through transformations: an experience report. In: Proc. Conf. Centre for Advanced Studies on Collaborative research (CASCON). IBM Press, p. 13.

Lam, M. S., Martin, M., Livshits, B., Whaley, J., 2008. Securing web applications with static and dynamic information flow tracking. In: PEPM '08: Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation. ACM, New York, NY, USA, pp. 3–12.

Lämmel, R., Nov. 2004a. Coupled software transformations (ext. abstract). In: Proceedings of International Workshop on Software Evolution Transformations (SETra). pp. 31–35.

Lämmel, R., 2004b. Transformations everywhere. Science of Computer ProgrammingThe guest editor's introduction to the SCP special issue on program transformation.

Lämmel, R., De Schutter, K., Mar. 2005. What does aspect-oriented programming mean to Cobol? In: Proceedings of Aspect-Oriented Software Development (AOSD 2005). ACM Press, pp. 99–110, 12 pages.

Lämmel, R., Lohmann, W., 2001. Format Evolution. In: Proc. 7th International Conference on Reverse Engineering for Information Systems (RETIS 2001). Vol. 155 of books@ocg.at. OCG, pp. 113–134.

Lämmel, R., van der Storm, T., 2009. Crossing the Rubicon of API Migration, unpublished manuscript. Available online `http://www.uni-koblenz.de/laemmel/apimigration/`.

Lämmel, R., Verhoef, C., December 2001. Semi-automatic Grammar Recovery. Software—Practice & Experience 31 (15), 1395–1438.

Lawley, M., Topor, R. W., 1994. A query language for eer schemas. In: Australasian Database Conference. pp. 292–304.

Liam O'Brien, Dennis Smith, G. L., 2005. Supporting migration to services using software architecture reconstruction. In: Proceedings IEEE International Workshop on Software Technology and Engineering Practice (STEP). pp. 81–91.

Lin, D.-Y., Neamtiu, I., August 2009. Collateral evolution of applications and databases. In: ERCIM Workshop on Software Evolution/International Workshop on Principles of Software Evolution. pp. 31–40.

Lohmann, W., Riedewald, G., 2003. Towards automatical migration of transformation rules after grammar extension. In: Proc. of 7th European Conference on Software Maintenance and Reengineering (CSMR'03). IEEE Computer Society Press, pp. 30–39.

Lopes, S., Petit, J.-M., Toumani, F., 1999. Discovery of "interesting" data dependencies from a workload of sql statements. In: Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery (PKDD'99). Springer-Verlag, London, UK, pp. 430–435.

Lucia, A. D., Francese, R., Scanniello, G., Tortora, G., Vitiello, N., 2006. A strategy and an eclipse based environment for the migration of legacy systems to multi-tier web-based architectures. In: Proc. Int'l Conf. Software Maintenance (ICSM). IEEE Computer Society, Washington, DC, USA, pp. 438–447.

Malton, A. J., August 2001. The software migration barbell. In: ASERC Workshop on Software Architecture.

Mantyla, M., 2003. Bad smells in software - a taxonomy and an empirical study. Ph.D. thesis, Helsinki University of Technology.

Marin, M., van Deursen, A., Moonen, L., 2004. Identifying aspects using fan-in analysis. In: Proc. Working Conf. Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, pp. 132–141.

Martin, J., Müller, H. A., 2001. Strategies for migration from C to Java. In: Proc. European Conf. Software Maintenance and Reengineering (CSMR). pp. 200–209.

Maule, A., Emmerich, W., Rosenblum, D. S., 2008. Impact analysis of database schema changes. In: Proceedings of the 30th international conference on Software engineering (ICSE'08. ACM Press, pp. 451–460.

Meier, A., 1995. Providing database migration tools - a practitioner's approach. In: Proceedings of International Conference on Very Large Data Bases (VLDB). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 635–641.

Meier, A., Dippold, R., Mercerat, J., Muriset, A., Untersinger, J.-C., Eckerlin, R., Ferrara, F., 1994. Hierarchical to relational database migration. IEEE Software 11 (3), 21–27.

Menhoudj, K., Ou-Halima, M., 1996. Migrating data-oriented applications to a relational database management system. In: Proceedings of the Third International Workshop on Advances in Databases and Information Systems. Moscow.

Mens, K., Kellens, A., Krinke, J., 2008. Pitfalls in aspect mining. In: WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering. IEEE Computer Society, Washington, DC, USA, pp. 113–122.

Merlo, E., Letarte, D., Antoniol, G., 2006. Insider and outsider threat-sensitive sql injection vulnerability analysis in php. In: Proc. Working Conf. Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, pp. 147–156.

Missaoui, R., Godin, R., Sahraoui, H., 1998. Migrating to an object-oriented databased using semantic clustering and transformation rules. Data Knowledge Engineering 27 (1), 97–113.

Ngo, M. N., Tan, H. B. K., 2008. Applying static analysis for automated extraction of database interactions in web applications. Inf. Softw. Technol. 50 (3), 160–175, to appear.

Papakonstantinou, Y., Gupta, A., Garcia-Molina, H., Ullman, J., 1995. A query translation scheme for rapid implementation of wrappers. In: Proceedings of International Conference on Declarative and Object-oriented Databases. pp. 161–186.

Papastefanatos, G., Anagnostou, F., Vassiliou, Y., Vassiliadis, P., 2008. Hecataeus: A what-if analysis tool for database schema evolution. In: Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08). IEEE Comp. Soc., pp. 326–328.

Petit, J.-M., Kouloumdjian, J., Boulicaut, J.-F., Toumani, F., 1994. Using queries to improve database reverse engineering. In: Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'94). Springer-Verlag, London, UK, pp. 369–386.

Petit, J.-M., Toumani, F., Kouloumdjian, J., 1995. Relational database reverse engineering: A method based on query analysis. Int. J. Cooperative Inf. Syst. 4 (2-3), 287–316.

Potts, C., 1993. Software-engineering research revisited. IEEE Softw. 10 (5), 19–28.

Rahm, E., Do, H., 2000. Data cleaning: Problems and current approaches. Data Engineering Bulletin 23, 3–13.

Reps, T., Teitelbaum, T., 1984. The synthesizer generator. SIGSOFT Softw. Eng. Notes 9 (3), 42–48.

Sahraoui, H. A., Lounis, H., Melo, W., Mili, H., 1999. A concept formation based approach to object identification in procedural code. Journal on Automated Software Engineering 6 (4), 387–410.

Sellink, A., Verhoef, C., 2000. Scaffolding for software renovation. In: CSMR '00: Proceedings of the Conference on Software Maintenance and Reengineering. IEEE Computer Society, Washington, DC, USA, p. 161.

Serrano, M. A., Carver, D. L., de Oca, C. M., 2002. Reengineering legacy systems for distributed environments. Systems and Software 64 (1), 37–55.

Shao, J., Liu, X., Fu, G., Embury, S. M., Gray, W. A., 2001. Querying data-intensive programs for data design. In: Proceedings of the 13th International Conference on Advanced Information Systems Engineering (CAiSE'01). Springer-Verlag, London, UK, pp. 203–218.

Signore, O., Loffredo, M., Gregori, M., Cima, M., 1994. Reconstruction of er schema from database applications: a cognitive approach. In: Proceedings of the 13th International Conference on the Entity-Relationship Approach (ER'94). Springer-Verlag, London, UK, pp. 387–402.

Sneed, H. M., 2000. Encapsulation of legacy software: A technique for reusing legacy software components. Annals of Software Engineering 9 (1-4), 293–313.

Sneed, H. M., 2006. Integrating legacy software into a service oriented architecture. In: Proc. European Conf. Software Maintenance and Reengineering (CSMR). IEEE Computer Society, Los Alamitos, CA, USA, pp. 3–14.

Stroulia, E., El-Ramly, M., Iglinski, P., Sorenson, P., 2003. User interface reverse engineering in support of interface migration to the web. Automated Software Engineering 10 (3), 271–301.

Tan, H. B. K., Ling, T. W., 1998. Correct program slicing of database operations. IEEE Software 15 (2), 105–112.

Tan, H. B. K., Ling, T. W., Goh, C. H., 2002. Exploring into programs for the recovery of data dependencies designed. IEEE Trans. Knowl. Data Eng. 14 (4), 825–835.

Tan, H. B. K., Zhao, Y., 2003. Automated elicitation of inclusion dependencies from the source code for database transactions. Journal of Software Maintenance 15 (6), 379–392.

Tennent, R., 1981. Principles of Programming Languages. Prentice Hall International Series in Computer Science.

Terekhov, A. A., Verhoef, C., 2000. The realities of language conversions. IEEE Software 17 (6), 111–124.

Thiran, P., Hainaut, J.-L., Houben, G.-J., Benslimane, D., October 2006. Wrapper-based evolution of legacy information systems. ACM Trans. Software Engineering and Methodology 15 (4), 329–359.

Tilley, S. R., Smith, D. B., 1995. Perspectives on legacy system reengineering. Tech. rep., Software Engineering Institute, Carnegie Mellon University.

Tonella, P., Ceccato, M., 2004. Aspect mining through the formal concept analysis of execution traces. In: Proc. Working Conf. Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, pp. 112–121.

Tourwe, T., Mens, K., September 2004. Mining aspectual views using formal concept analysis. In: Proc. Workshop Source Code Analysis and Manipulation (SCAM). pp. 97–106.

van Den Brand, M., Klint, P., Vinju, J. J., 2003. Term rewriting with traversal functions. ACM Transaction on Software Engineering and Methodology 12 (2), 152–190.

van den Brand, M., Moreau, P.-E., Vinju, J. J., 2003. Environments for term rewriting engines for free! In: Proceedings of the 14th International Conference on Rewriting Techniques and Applications (RTA'03). pp. 424–435.

van den Brand, M., van Deursen, A., Heering, J., de Jong, H., de Jonge, M., Kuipers, T., Klint, P., Moonen, L., Olivier, P., Scheerder, J., Vinju, J., Visser, E., Visser, J., 2001. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In: Wilhelm, R. (Ed.), Compiler Construction (CC '01). Vol. 2027 of Lecture Notes in Computer Science. Springer-Verlag, pp. 365–370.

van den Brand, M., van Deursen, A., Klint, P., Klusener, S., van der Meulen, E., 1996. Industrial applications of asf+sdf. In: Wirsing, M., Nivat, M. (Eds.), Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology (AMAST'96). Vol. 1101 of Lecture Notes in Computer Science. Springer, pp. 9–18.

van den Brand, M., Vinju, J. J., 2000. Rewriting with layout. In: Kirchner, C., Dershowitz, N. (Eds.), Proceedings of the First International Workshop on Rule-Based Programming (RULE'2000).

van den Brand, M. G. J., Bruntink, M., Economopoulos, G. R., de Jong, H. A., Klint, P., Kooiker, A. T., van der Storm, T., Vinju, J. J., 2007. Using the meta-environment for maintenance and renovation. In: Krikhaar, R. L., Verhoef, C., Lucca, G. A. D. (Eds.), Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07). IEEE Computer Society, pp. 331–332.

van den Brand, M. G. J., Kooiker, A. T., Vinju, J. J., Veerman, N. P., 2006. A language independent framework for context-sensitive formatting. In: Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR'06). IEEE Computer Society, pp. 103–112.

van den Brink, H., van der Leek, R., Visser, J., 2007. Quality assessment for embedded sql. In: Proceedings of the 7th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'07). IEEE Computer Society, pp. 163–170.

van Deursen, A., Kuipers, T., 1998. Rapid system understanding: Two cobol case studies. In: Int'l Workshop on Program Comprehension. IEEE Comp. Soc., pp. 90–98.

van Deursen, A., Kuipers, T., 1999. Identifying objects using cluster and concept analysis. In: Proc. Int'l Conf. Software Engineering (ICSE). IEEE Computer Society Press, Los Alamitos, CA, USA, pp. 246–255.

van Emden, E., Moonen, L., 2002. Java quality assurance by detecting code smells. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02). IEEE Computer Society Press.

Veerman, N., 2004. Revitalizing modifiability of legacy assets. Software Maintenance and Evolution: Research and Practice 16 (4–5), 219–254.

Veerman, N., 2006. Automated mass maintenance of a software portfolio. Science of Computer Programming 62, 287–317.

Veerman, N., 2007. Automated mass maintenance of software assets. Ph.D. thesis, Vrije Universiteit, Amsterdam.

Vermolen, S., Visser, E., 2008. Heterogeneous coupled evolution of software languages. In: MoDELS '08: Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. Springer-Verlag, Berlin, Heidelberg, pp. 630–644.

Visaggio, G., 2001. Ageing of a data-intensive legacy system: symptoms and remedies. Journal of Software Maintenance 13 (5), 281–308.

Visser, J., 2008. Coupled transformation of schemas, documents, queries, and constraints. Electr. Notes Theor. Comput. Sci. 200 (3), 3–23.

Warren, I., 1999. The Renaissance of Legacy Systems: Method Support for Software-System Evolution. Springer-Verlag, Secaucus, NJ, USA.

Waters, R. C., 1988. Program translation via abstraction and reimplementation. IEEE Computer Society Trans. Software Engineering 14 (8), 1207–1228.

Weiser, M., 1984. Program slicing. IEEE Transactions on Software Engineering 10 (4), 352–357.

Wiederhold, G., 1995. Modeling and system maintenance. In: Proceedings of the International Conference on Object-Oriented and Entity-Relationship Modeling. Berlin, pp. 1–20.

Willmor, D., Embury, S. M., Shao, J., 2004. Program slicing in the presence of a database state. In: ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance. IEEE Computer Society, Washington, DC, USA, pp. 448–452.

Wu, B., Lawless, D., Bisbal, J., Grimson, J., Wad, V., O'Sullivan, D., Richardson, R., 1997. Legacy system migration: A legacy data migration engine. In: Experts, E. C. C. (Ed.), Proceedings of the 17th International Database Conference (DATASEM '97). pp. 129–138.

Wu, B., Lawless, D., Bisbal, J., Richardson, R., Grimson, J., Wade, V., O'Sullivan, D., September 1997. The butterfly methodology: A gateway-free approach for migrating legacy information systems. In: Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems. Italy, pp. 200–205.

Yang, H., Chu, W. C., 1999. Acquisition of entity relationship models for maintenance-dealing with data intensive programs in a transformation system. J. Inf. Sci. Eng. 15 (2), 173–198.

Yasumatsu, K., Doi, N., 1995. Spice: A system for translating smalltalk programs into a C environment. IEEE Computer Society Trans. Software Engineering 21 (11), 902–912.

Yeh, A. S., Harris, D. R., Reubenstein, H. B., 1995. Recovering abstract data types and object instances from a conventional procedural language. In: Proc. Working Conf. Reverse Engineering (WCRE). IEEE Computer Society, Washington, DC, USA, p. 227.

Zou, Y., Kontogiannis, K., 2001. A framework for migrating procedural code to object-oriented platforms. In: Proceedings IEEE Asia-Pacific Software Engineering Conf. (APSEC). IEEE Computer Society, Los Alamitos, CA, USA, pp. 408–418.

Appendix A

COBOL File Handling Statements

A.1 `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 right after the last of the file for *writing*.

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

A.2 `CLOSE` statement

Syntax `CLOSE` *file-name*

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

A.3 `START` statement

Syntax

```
START file-name KEY IS relational-operator record-key
      INVALID KEY statements-1
      NOT INVALID KEY 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 read the file sequentially starting from that record. The INVALID KEY phrase indicates the statements to be executed if a record with the specified key value cannot be found. Conversely, the optional NOT INVALID KEY phrase specifies how the program behaves if such a record is found. The INVALID/NOT INVALID clauses can also be associated with the READ, WRITE, REWRITE and DELETE statements, in the case of indexed or relative files.

A.4 READ statement

There exist two kinds of READ statements in COBOL, depending on the access mode of the file to read.

A.4.1 Format 1 (sequential access mode)

Syntax

```
READ file-name NEXT [INTO identifier]
     AT END statements-1
     NOT AT END 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 execute statements if no end of file is encountered, thus, if a record is read.

If the INTO clause is specified, the resulting record is moved to the specified *identifier*. If the INTO clause if omitted, the resulting record is processed directly in the record area (i.e., in the buffer).

Reference key By default, the reference key of a file is the primary key of the corresponding record type. In the case of an indexed file, the reference key may vary during the execution of the program. Indeed, the execution of a START or a random READ statement causes the specified access key to become the reference key. Note that a static analysis of the program does not allow to figure out which is the reference key of a file at a given point of the program. For instance, the same

```
1      READ-ORD-CODE.
2        START ORDERS KEY IS > ORD-CODE
3          INVALID KEY
4            MOVE 0 TO END-FILE
5          NOT INVALID KEY
6            MOVE 1 TO END-FILE.
7        PERFORM READ-ORD-NEXT
8          UNTIL END-FILE = 0.
9
10     READ-ORD-DATE.
11       START ORDERS KEY IS > ORD-DATE
12          INVALID KEY
13           MOVE 0 TO END-FILE
14         NOT INVALID KEY
15           MOVE 1 TO END-FILE.
16        PERFORM READ-ORD-NEXT
17          UNTIL END-FILE = 0.
18
19     READ-ORD-NEXT.
20        READ ORDERS NEXT
21          AT END
22            MOVE 0 TO END-FILE
23          NOT AT END
24            ...
```

Figure A.1: Multiple possible reference keys for a single READ NEXT statement.

READ NEXT statement can be reached from two different START statements during the execution of the program. Figure A.1 shows such an example corresponding to file ORDERS declared in Figure 7.1. The reference key used by the sequential read statement located at line 20 can be either ORD-CODE or ORD-DATE, depending on the PERFORM statement from which the READ-ORD-NEXT procedure is called (line 7 or line 16).

Format 2 (random access mode)

Syntax

READ *file-name* [INTO *identifier*] [KEY IS *record-key*]
    INVALID KEY *statements-1*
    NOT INVALID KEY *statements-2*
END-READ

Effect If the access mode is RANDOM, the programmer has to supply a key for retrieving a record. If the KEY phrase is omitted, COBOL assumes the RECORD KEY of the file is used. The INVALID KEY phrase declares the statements to be executed if a record with the specified key cannot be found. Converserly, the optional NOT INVALID KEY phrase specifies the behaviour of the program if such a record is retrieved.

A.5 WRITE statement

The format of the `WRITE` statement depends on the file access mode.

Format 1 (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. If the `FROM` phrase is omitted, the inserted record is taken from the corresponding buffer. If the `FROM` phrase is specified, the record to be written is taken from the program variable *identifier*.

Format 2 (random access mode)

Syntax

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

Effect In the `RANDOM` access mode, the `INVALID KEY` phrase 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 successfully inserted in the file.

A.6 REWRITE statement

Format 1 (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 of the current record buffer (or the content of *identifier*). In a `SEQUENTIAL` access mode, a record must be read before it can be rewritten.

Format 2 (random access mode)

Syntax

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

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 successfully rewritten.

A.7 DELETE statement

The DELETE statement allows to delete a record.

Format 1 (sequential access mode)

Syntax DELETE *file-name* [RECORD]

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

Format 2 (random access mode)

Syntax

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

Effect For the RANDOM or DYNAMIC access modes, a value has first to be assigned to the record key, in order to indicate the record to be deleted. If the provided record key is invalid, the INVALID KEY phrase executes predefined statements. The NOT INVALID KEY phrase specifies the statements to execute if a record is successfully rewritten.

Appendix B

LDA Language: Syntax and Partial Semantics

B.1 Concrete syntax

```
LDA-program-head  : "program" LDA-identifier "." Schema-declaration  ;

Schema-declaration  : "schema" LDA-literal-value ";"  ;

LDA-variable-declaration  : LDA-type ":" {LDA-variable-name","}+ ";"  ;

LDA-compound-id  : LDA-identifier  | LDA-identifier "." LDA-compound-id ;

LDA-type  : LDA-entity-type-name
          | LDA-entity-type-name "." LDA-compound-id
          | "string"
          | "integer"
          | "boolean"
          | LDA-type "(" NatCon ")"  ;

LDA-declarations  : LDA-variable-declaration+  ;

LDA-program-body  : "begin" {LDA-statement ";"}+ "end" "."  ;

LDA-program  : LDA-program-head LDA-declarations? LDA-program-body  ;

LDA-sequence  : {LDA-statement ";"}+  ;

LDA-statement  : LDA-assigment
               | LDA-if-statement
               | LDA-while-loop
               | LDA-for-loop
               | LDA-delete
               | LDA-create
               | LDA-modify
               | LDA-input
               | LDA-print  ;

LDA-assigment  : LDA-variable-reference Assign-symbol LDA-expression  ;

LDA-if-statement  : "if" LDA-condition "then" LDA-sequence Else-phrase? "endif"  ;

Else-phrase  : "else" LDA-sequence  ;
```

```
LDA-while-loop  : "while" LDA-condition "do" LDA-sequence "endwhile"  ;

LDA-for-loop  : "for" LDA-variable-reference Assign-symbol Seq-order? LDA-expression
                "do" LDA-sequence "endfor"
              | "for" LDA-variable-reference "in" Seq-order  "do" LDA-sequence "endfor"  ;

LDA-delete  : "delete" LDA-variable-reference LDA-condition?  ;

LDA-modify  : "modify" LDA-variable-reference LDA-condition  ;

LDA-create  : "create" LDA-variable-reference Assign-symbol What-is-created  ;

LDA-input  : "input" "(" {LDA-variable-reference ","}+ ")"  ;

LDA-print  : "print" "(" {LDA-expression ","}+ ")"  ;

Seq-order  : Integer
           | Integer-interval  ;

Integer-interval  : Integer ".." Integer  ;

Assign-symbol  : ":" "="  ;

LDA-condition  : LDA-simple-condition
               | LDA-condition "and" LDA-condition  ;
               | "(" LDA-condition ")";
               | Not-kw "(" LDA-condition ")";

LDA-simple-condition  : LDA-expression LDA-relop LDA-expression
                      | LDA-expression  ;

Entity-selection  : LDA-entity-type-name LDA-condition  ;

What-is-created  : Entity-selection  ;

LDA-expression  : LDA-term
                | "{" {Entity-selection ","}+ "}"
                | Integer LDA-expression
                | LDA-expression Plus-sign LDA-expression
                | LDA-expression Minus-sign LDA-expression
                | "(" LDA-expression ")"
                | Entity-selection
                | LDA-rel-type-name ":" LDA-expression  ;

LDA-term  : ":" LDA-item-name
          | LDA-item-name
          | LDA-literal-value
          | LDA-variable-reference
          | Integer
          | Boolean  ;

Not-kw  : "not"  ;

LDA-equal  : "="  ;

LDA-relop  : Not-kw LDA-equal
           | "in"
           | LDA-equal
           | ">"
           | "<"
           | "<>"
           | "<="
           | ">="  ;

Plus-sign  : "+"  ;
```

```
Minus-sign  : "-"  ;

LDA-rel-type-name  : LDA-identifier  ;

LDA-entity-type-name  : LDA-identifier  ;

LDA-item-name  : LDA-identifier
               | LDA-identifier OccurenceNumber? "." LDA-item-name  ;

LDA-variable-reference  : LDA-variable-name
                   | LDA-variable-name OccurenceNumber? "." LDA-item-name
                   | LDA-variable-name OccurenceNumber? "." LDA-variable-reference  ;

OccurenceNumber  : "[" NatCon "]"
                 | "[" LDA-variable-reference "]"  ;

Integer  : Natural
         | "+" Natural
         | "-" Natural
         | Integer "+" Integer
         | Integer "-" Integer
         | Integer "*" Integer
         | "max" "(" Integer "," Integer ")"
         | "(" Integer ")"  ;

Natural  : NatCon
         | Natural "-/" Natural
         | "(" Natural ")"
         | Natural "-//" Natural
         | Natural ">-" Natural  ;

Boolean  : Integer ">" Integer
         | Integer ">=" Integer
         | Integer "<" Integer
         | Integer "<=" Integer
         | "gt" "(" Natural "," Natural ")"
         | BoolCon
         | Boolean "|" Boolean
         | Boolean "&" Boolean
         | "not" "(" Boolean ")"
         | "(" Boolean ")"
         | Boolean "&" Boolean
         | Boolean "|" Boolean  ;

BoolCon  : "true"  | "false"  ;
```

B.2 Semantics

B.2.1 Syntactic Domains

Ide	identifier
Exp	expressions
$Stat$	statements
$BCond$	boolean condition
$ECond$	entity selection condition

B.2.2 Semantic Domains

$\mathbb{B}$	booleans
$\mathbb{Z}$	integers
$\mathbb{R}$	database references
$v \in \mathbb{V} = \mathbb{B} + \mathbb{Z} + \mathbb{R}$	basic values
$g \in \mathbb{G} = \mathbb{V} + \mathbb{C}$	GER values
$\mathbb{C} = Ide \to 2^{\mathbb{G}}$	complex GER values
$s \in \mathbb{S} = Ide \to \mathbb{V} + null$	store
$d \in \mathbb{D} = \mathbb{D}_E \times \mathbb{D}_R$	database state
$d_E \in \mathbb{D}_E = \mathbb{R} \to \mathbb{G}$	database state (entity type instances)
$d_R \in \mathbb{D}_R = (\mathbb{R} \times Ide) \to 2^{\mathbb{R}}$	database state (relationship type instances)
$o \in \mathbb{O} = (\mathbb{S} \times \mathbb{D}) + \{error\}$	statement result

B.2.3 Semantic Functions

$\mathscr{E} : Exp \to \mathbb{S} \to \mathbb{D} \to \mathbb{G}$	evaluating expressions
$\mathscr{S} : Stat \to \mathbb{S} \to \mathbb{D} \to \mathbb{O}$	evaluating statements

B.2.4 Notations

We will use the following notations, some of which are inspired from Tennent (1981).

(a) We define a ternary selection operation ’$\cdot \to \cdot , \cdot$ ’ as follows:

$$e \to x_1, x_2 == \begin{cases} x_1, \text{ if } e = true \\ x_2, \text{ if } e = false \\ error, \text{ if } e \text{ is not a boolean expression.} \end{cases}$$

(b) We define a ternary operation ’$\cdot \to \cdot , \cdot$ ’ for *perturbing* a function as follows: $f[x \to y]$ is the function that is like f except that argument x is mapped into y.

(c) We will note $d(E) = \{r_1 : \mathbb{R}\}$ the set of database references in database state d corresponding to instances of entity type E.

(e) For concision, we will note $\mathscr{E}_{s,d}[\![exp]\!] = \mathscr{E}[\![exp]\!](s)(d)$ the application of the semantic fonction $\mathscr{E}$ on expression exp with respect to store s and database state d (and similarly for function $\mathscr{S}$).

(f) We will note $type(Var)$ the type of variable Var.

B.2.5 Semantics of Expressions (for record selection)

Attribute-based condition

$$\mathscr{E}_{s,d}[\![E(:A = exp)]\!] \quad = \quad \{r \in \mathbb{R} : r \in d(E) \wedge d_E(r)(A) = \mathscr{E}_{s,d}[\![exp]\!]\}$$

Relationship-based condition

$$\begin{aligned}\mathscr{E}_{s,d}[\![E_1(R : E_2(cond))]\!] \quad &= \quad \{r_1 \in \mathbb{R} : r_1 \in d(E_1) \wedge \\ &\quad\ \exists r_2 \in \mathbb{R} : r_2 \in d_R(r_1, R) \wedge r_2 \in \mathscr{E}_{s,d}[\![E_2(cond)]\!]\}\end{aligned}$$

Compound condition

$$\mathscr{E}_{s,d}[\![E(cond_1 \ and \ cond_2)]\!] = \{r \in \mathbb{R} : r \in \mathscr{E}_{s,d}[\![E(cond_1)]\!] \cap \mathscr{E}_{s,d}[\![E(cond_2)]\!]\}$$

B.2.6 Semantics of Statements (for record manipulation)

Assignment (of record selection expression)

$$\begin{aligned}\mathscr{S}_{s,d}[\![Var \ \texttt{:=} \ E(cond)]\!] = \mathscr{E}_{s,d}[\![E(cond)]\!] \neq \emptyset \rightarrow (s[Var \mapsto r], d), (s[Var \mapsto null], d) \\ \text{where} \quad r \in \mathscr{E}_{s,d}[\![E(cond)]\!]\end{aligned}$$

Conditional statement (with record selection expression as condition)

$$\begin{aligned}&\mathscr{S}_{s,d}[\![\texttt{if} \ E(cond) \ \texttt{then} \ stats_1 \ \texttt{else} \ Stats_2]\!] \\ &\quad = \quad \mathscr{E}_{s,d}[\![E(cond)]\!] \neq \emptyset \rightarrow \mathscr{S}_{s,d}[\![Stats_1]\!], \mathscr{S}_{s,d}[\![Stats_2]\!]\end{aligned}$$

For loop (based on record selection expression)

$$\begin{aligned}\mathscr{S}_{s,d}[\![\texttt{for} \ Var \ \texttt{:=} \ E(cond) \ \texttt{do} \ Stats]\!] \quad = \quad & \mathscr{E}_{s,d}[\![E(cond)]\!] \neq \emptyset \rightarrow (s'', d''), (s, d) \\ \text{where} \quad & \text{let } X = \mathscr{E}_{s,d}[\![E(cond)]\!] \\ & \text{let } e \in X \\ & \text{let } Y = X \setminus \{e\} \\ & (s', d') = \mathscr{S}_{s[Var \mapsto e],d}[\![Stats]\!] \\ & (s'', d'') = \mathscr{S}_{s',d'}[\![\texttt{for} \ Var\texttt{:=} \ Y \ \texttt{do} \ Stats]\!]\end{aligned}$$

Create primitive

$$\begin{aligned}\mathscr{S}_{s,d}[\![\texttt{create} \ Var\texttt{:=}E(cond)]\!] \quad = \quad & (s', d') \\ \text{where} \quad & \text{let } r \in \mathbb{R} : d(r) = null \\ & d'(E) = d(E) \cup \{r\} \\ & r \in \mathscr{E}_{s',d'}[\![E(cond)]\!] \\ & s' = s[Var \mapsto r]\end{aligned}$$

Delete primitive

$$\begin{aligned}\mathscr{S}_{s,d}[\![\texttt{delete} \ Var(cond)]\!] \quad = \quad & r \in \mathscr{E}_{s,d}[\![E(cond)]\!] \rightarrow (s', d'), (s, d) \\ \text{where} \quad & r = s(Var) \\ & E = type(Var) \\ & s' = s[Var \mapsto null] \\ & d'(E) = d(E) \setminus \{r\}]\end{aligned}$$

Update primitive

$$\begin{array}{lll} \mathscr{S}_{s,d}[\![\texttt{update}\, Var(cond)]\!] & = & (s, d') \\ \text{where} \quad \text{let } E = type(Var) & & \\ \qquad\qquad s(Var) \in \mathscr{E}_{s,d'}[\![E(cond)]\!] & & \end{array}$$