

RESEARCH OUTPUTS / RÉSULTATS DE RECHERCHE

Integration of Legacy and Heterogeneous Databases

Thiran, Philippe; Hainaut, Jean-Luc

Publication date:
2002

[Link to publication](#)

Citation for pulished version (HARVARD):

Thiran, P & Hainaut, J-L 2002, *Integration of Legacy and Heterogeneous Databases*. Institut d'Informatique - LIBD, Namur.

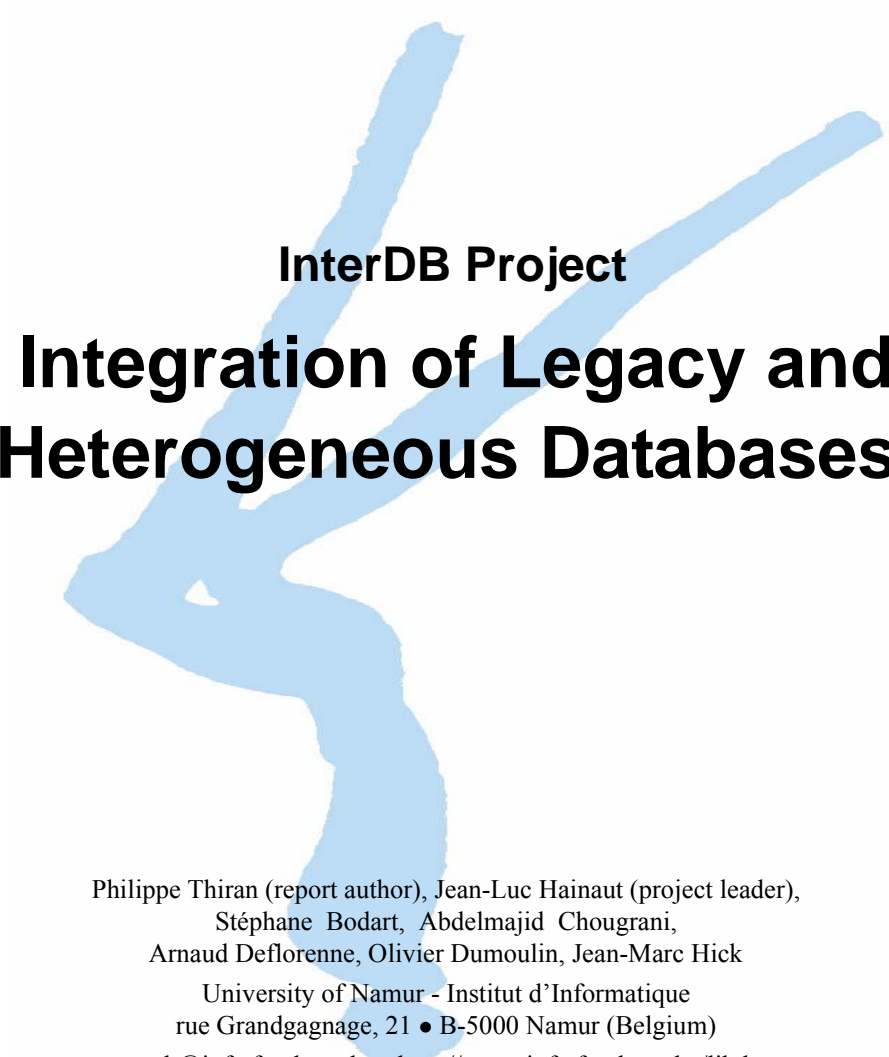
General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.



InterDB Project

Integration of Legacy and Heterogeneous Databases

Philippe Thiran (report author), Jean-Luc Hainaut (project leader),
Stéphane Bodart, Abdelmajid Chougrani,
Arnaud Deflorenne, Olivier Dumoulin, Jean-Marc Hick
University of Namur - Institut d'Informatique
rue Grandgagnage, 21 • B-5000 Namur (Belgium)
pth@info.fundp.ac.be - <http://www.info.fundp.ac.be/libd>

Credits

This text is based on the results of the *InterDB project* supported by the Belgian *Région Wallonne* (Direction Générale des Télécommunications, de la Recherche et de l'Energie).

InterDB Project

The InterDB Project is a research, development and technology transfer programme in data integration engineering, undertaken at the Institute of Informatics of the University of Namur since September 1995. The InterDB project is dedicated to the integration and the interoperability of heterogeneous and distributed information systems. It directly profits from research and development results that have been developed in the DB-MAIN research project.

Research Aspects of InterDB

Though some aspects of the databases integration can now be considered as fairly well understood, some complex problems and processes still are unsolved at the present time, despite an increasing interest of the scientific community. Such is the case of semantic recovery and semantic integration. The InterDB project is intended to contribute to solving these problems by concentrating on their database aspects first.

The technical material of InterDB has been described in about ten scientific papers published in the main international conferences and journals since 1995. The basic principles have been described in the three papers:

- Ph. Thiran, J-L. Hainaut, S. Bodart, A. Deflorenne, J-M. Hick, "Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE Support: the InterDB Approach" in *Proceedings of CoopIS'98*, IEEE, New York, August 1998.
- J-L. Hainaut, Ph. Thiran, J-M. Hick, S. Bodart, A. Deflorenne, "Methodology and CASE tools for the development of federated databases", *International Journal of Cooperative Information Systems*, 8(2-3), pp. 169-194, World Scientific, June and September 1999.
- Ph. Thiran, J-L. Hainaut, "Wrapper Development for Legacy Data Reuse", in *Proceedings of WCRE'01*, IEEE, Stuttgart, October 2001.

The InterDB approach of the database integration has been presented in several main international conferences:

- Main conferences: IFCIS International Conference on Cooperative Information Systems, 1998 - New York City; Database and Expert System Applications Conference, 1999 - Florence; International Workshop on Engineering Federated Information Systems, 2000 and 2001 - Dublin and Berlin; and IEEE Working Conference on Reverse Engineering, 2001 - Stuttgart.

- Main foreign universities: EPFL, Magdeburg Universität, Laboratoire d'Informatique de Paris VI, Université de Lyon I, Université de Lausanne and Universiteit van Tilburg.

Development Aspects of InterDB

The scientific material that is developed in research activities is implemented as software and CASE tool components. They transform the research results into practical methods and techniques that can be most helpful for practitioners. In that way, the InterDB materials have been used by two joint projects: the *Data Migration Project* [Delcroix, 2001] and the *Administrative Database Integration Project* (with the City Council of Namur).

Technology Transfer of InterDB

As an academic institution, the University of Namur, and particularly the Institute of Informatics, is strongly committed to making knowledge available to as large as possible an audience. Accordingly, the InterDB results are translated into educational materials such as case studies, lectures and training seminars, mainly intended to the industrial partners.

InterDB Team

The InterDB team comprises research associates: Stéphane Bodart, Abdelmajid Chougrani, Arnaud Deflorenne, Olivier Dumoulin, Jean-Marc Hick and Philippe Thiran. The management and scientific direction of the InterDB project is carried out by professeur Jean-Luc Hainaut.

The first materials of the InterDB project have been tested and improved by several students: Renaud Denis [Denis, 2002], Bernard Noël [Noel, 2001] and Sybille Radulescu [Radulescu, 2001].

Acknowledgments

We thank Jean-Luc Hainaut for his support, his supervision and his very useful advices. Without him, nothing would have been possible. We also thank Jean-Marc Hick for his rereading of this report.

À Mon Hélène
Philippe

Summary

Accessing and managing data from several existing independent databases pose complex problems that can be classified into platform, DMS, location and semantic levels. The platform level copes with the fact that databases reside on different brands of hardware, under different operating systems, and interacting through various network protocols. Leveling these differences leads to platform independence. DMS level independence allows programmers to ignore the technical details of data implementation in a definite family of models. It can also hide the model that the DMS implements by providing a more abstract model. Location independence isolates the user from knowing where the data reside. Finally, semantic level independence solves the problem of multiple, replicated and conflicting representations of similar facts.

The InterDB project proposes a general architecture, a methodology and a CASE environment intended to address the problem of providing users and programmers with an abstract interface to independent, heterogeneous and distributed databases.

Architecture. The architecture comprises a hierarchy of mediators that dynamically transform actual data into a virtual homogeneous database. Each layer of mediators provides a certain kind of independence. DMS independence is provided by wrappers dedicated to each database. Location and semantic independence's are ensured by a mediator. Finally, platform independence is ensured by both the wrappers and ad hoc middleware such as commercial ORB.

Methodology. Such an architecture involves controlling complex mappings. The problem is complicated by the fact that the databases have been developed independently, and naturally suffer from sever problems of replication and semantic conflicts. In addition, most legacy databases have no documentation any more, just like programs. Recovering the conceptual schemas form an existing database is the main goal of database reverse engineering, an important software engineering that can now be considered mature. Solving the syntactic and semantic conflicts of independent schemas has long been studied in the database realm. However, coping with conceptual schemas form populated databases brings new problems. A complete methodology, encompassing conceptual schema recovery and database integration is provided to practitioners.

Case support. Deriving a common, abstract and conflict-free image of independent databases and defining the mappings between the specification layers are complex tasks. Building the wrappers and the mediators are also two complex and error-prone activities. All these processes are supported by the DB-MAIN CASE tool. This graphical, repository-based, software engineering environment includes, among others, a sophisticated reverse engineering toolkit, schema mapping specification facilities and a generator development environment.

Table of Contents

Chapter 1 - Introduction

Part I: Generic Integration Framework

Chapter 2 - Integration Architecture

Chapter 3 - Generic Data Model

Chapter 4 - Mapping Definition

Part II: Wrapper Technology

Chapter 5 - Wrapper Architecture

Chapter 6 - Wrapper Development (Methodology)

Chapter 7 - Wrapper Development Support

Part III: Mediator Technology

Chapter 8 - Mediator Architecture

Chapter 9 - Mediator Development (Methodology)

Chapter 10 - Mediator Development Support

References

The InterDB Project (1995-2002) has been supported by the Belgian
Région Wallonne (Direction Générale des Télécommunications, de la
Recherche et de l'Energie) • Report Edition 1.1 • November 2002



Detailed Table of Contents

Chapter 1

Introduction	1
1.1 Introduction	1
1.2 Problem and Context of InterDB.....	2
1.2.1 Legacy Data Systems.....	2
1.2.2 Distribution	2
1.2.3 Autonomy	3
1.2.4 Heterogeneity.....	3
1.2.5 Mediation	4
1.2.6 Mediation and Legacy Databases	4
1.2.7 Database Integration	6
1.3 Scope and Overview.....	8
1.3.1 Scope.....	8
1.3.2 Overview.....	8

Part I - Generic Integration Framework

Chapter 2

Integration Architecture.....	13
2.1 Introduction	13
2.2 Overview of integration architectures	13
2.2.1 Global Schema Systems.....	15
2.2.2 Multidatabase Languages	15
2.2.3 Federated Architecture.....	16
2.3 InterDB Architecture	18
2.3.1 Hierarchy Architecture	18
2.3.2 Component Architecture.....	19

Chapter 3

Generic Data Model.....	21
3.1 Introduction	21
3.2 Generic Data Model	22
3.2.1 Main concepts	22
3.2.2 Meta concepts	24
3.2.3 Model specialization	25

3.3 Federation Data Models	26
3.3.1 Legacy Data Models	26
3.3.2 Canonical Data Models	28
3.3.3 Object-oriented Model	30

Chapter 4

Mapping Definition	33
4.1 Introduction	33
4.2 Mapping Baselines	33
4.3 Definition	34
4.3.1 Reversibility	35
4.3.2 Structural Analysis of a Transformation	36
4.3.3 Signature of a Transformation	36
4.3.4 Schema Transformation Sequence	37
4.3.5 Schema Integration	38
4.4 Some Popular Transformations	39
4.5 Transformation History	40
4.5.1 Structure of a History	40
4.5.2 History Subset	41
4.5.3 Independent Histories	42
4.5.4 Equivalent Histories	42
4.5.5 Minimal History	43
4.6 Model Transformation	44

Part II - Wrapper Technology

Chapter 5

Wrapper Architecture	49
5.1 Introduction	49
5.2 Legacy Data Wrapper Definition	51
5.2.1 Overview	51
5.2.2 Definition	51
5.2.3 Functionality	51
5.2.4 Legacy Issues	52
5.2.5 Motivations and Objectives	54
5.3 Architecture	55
5.3.1 General Framework	55
5.3.2 Wrapper Interface	57
5.4 Wrapper Services	60
5.4.1 Query Analysis	61

5.4.2 Error Reporting	62
5.4.3 Functionality Emulation	62
5.4.4 Query Processing	66
5.4.5 Semantic Integrity Control.....	70
5.5 InterDB Prototypes	74
5.5.1 Logical Wrapper	76
5.5.2 Object Wrapper.....	80

Chapter 6

Wrapper Development 83

6.1 Introduction	83
6.2 Wrapper Development Baselines	84
6.2.1 Observations	84
6.2.2 Wrapper Dimensions	84
6.2.3 Wrapper Models and Schemas	86
6.2.4 Wrapper Mapping	87
6.2.5 Logical Wrapper Architecture	88
6.3 Methodology for Logical Wrapper Development	91
6.3.1 Overview	91
6.3.2 Development Baselines.....	92
6.3.3 Wrapper Definition	92
6.3.4 Logical Wrapper Definition.....	94
6.3.5 Wrapper Packaging.....	95
6.3.6 Generator Application and Maintenance	95
6.4 Methodology for Instance Wrapper Generation	95
6.4.1 Data-centered Reverse Engineering.....	96
6.4.2 Catalog of Implicit Constraints and Constructs.....	100
6.4.3 Wrapper Schema Definition	104
6.4.4 Mapping Definition.....	105
6.5 Methodology for Upper Wrapper Development	106

Chapter 7

Wrapper Development Support 107

7.1 Introduction	107
7.2 CASE Tool Requirements	108
7.2.1 General Support	108
7.2.2 Support of the Data-centered Reverse Engineering.....	108
7.2.3 Support of the Mapping Definition.....	109
7.3 DB-MAIN.....	109
7.3.1 User Interface.....	109
7.3.2 DB-MAIN Specification Model and Repository	110

7.3.3 Voyager 2.....	111
7.3.4 Transformation Toolkit.....	112
7.3.5 Text Analysis and Processing.....	113
7.3.6 Assistants	114
7.3.7 History	118
7.4 InterDB Tools.....	119
7.4.1 History Analyzer.....	121
7.4.2 Wrapper Encoders.....	123

Part III - Mediator Technology

Chapter 8

Mediator Architecture.....	127
8.1 Introduction	127
8.2 Mediator Definition	127
8.3 Architecture	129
8.3.1 General Framework	129
8.3.2 Mediator Interface.....	130
8.4 Mediator Services.....	134
8.4.1 Query Analysis	134
8.4.2 Query Processing	135
8.4.3 Security Management	138
8.4.4 Global Semantic Integrity Management.....	139
8.4.5 Transaction Management.....	140
8.5 InterDB Prototype	143
8.5.1 Mediator Architecture.....	143
8.5.2 Object Mediator and DB-MAIN Repository	144
8.5.3 Algorithm Principle of the Object Mediator.....	147

Chapter 9

Mediator Development	149
9.1 Introduction	149
9.2 Framework for Schema Integration.....	150
9.2.1 Integration Strategies	151
9.2.2 Pre-integration	151
9.2.3 Correspondence Identification.....	153
9.2.4 Schema Integration	153
9.2.5 Mapping Definition.....	154
9.3 Schema Integration Issues	155
9.3.1 Interschema Correspondences	155

9.3.2 Interschema Conflicts	158
9.4 InterDB Approach	160
9.4.1 InterDB Principles	160
9.4.2 Practical InterDB Methodologies	164

Chapter 10

Mediator Development Support 167

10.1 Introduction	167
10.2 CASE Tool Requirements	168
10.3 DB-MAIN.....	169
10.3.1 DB-MAIN Repository	169
10.3.2 Integration Assistants.....	170
10.3.3 History	174
10.4 InterDB Tools	175
10.4.1 InterDB Extension of the DB-MAIN Repository	175
10.4.2 History Analyzer.....	181
10.4.3 Java Access to the DB-MAIN Repository	182

References 185

Introduction

In which the reader is introduced to integration of legacy databases by first giving its main issues. The terms legacy, autonomy, heterogeneity and mediation are introduced, and a short overview of the InterDB approach is given.

1.1 Introduction

Most large organizations maintain their data in many distinct independent databases that have been developed at different times on different platforms and DMS (Data Management Systems).

The new economic challenges force enterprises to integrate their functions and therefore their information systems including the databases they are based on. In most cases, these databases cannot be replaced with a unique system, nor even reengineered due to the high financial and organizational costs of such a restructuring. Hence the need for interoperation frameworks that allow the databases to be accessed by users and application programs as if they were a unique homogeneous and consistent database, through an architecture called federated databases.

1.2 Problem and Context of InterDB

1.2.1 Legacy Data Systems

The presence of legacy data systems is one of the major obstacles in the use of integrated information.

A legacy IS is any IS that significantly resists modifications and changes. Typically, a legacy IS is big, with millions of lines of code, and more than 10 years old. [Brodie 1995]

Legacy data systems are very large. They are written in old programming language like COBOL or PL/1. Such systems are usually mission critical to the day-to-day operation of operation of corporations and are thus very valuable from a business point of view. However, they are inflexible in nature and expensive to maintain or to change [Bouguettaya, 1998].

Legacy data systems must be kept as they are for several reasons [Umar, 1997]. First, they provide vital services that are very risky to disrupt. Second, many users and support staff are trained on how to operate these systems and to use them. Third, many legacy systems are very reliable and perform very well, contrary to the common belief. Fourth, the administrative support of mainframe-based legacy applications (e.g., backup/recovery, change management, security) has matured over the years. Finally, there is some emotional attachment to legacy applications among senior staff because these systems have survived through years of fundamental changes in business practices and technologies.

However, something must be done about these systems. First, legacy applications are becoming increasingly expensive to maintain and operate (it takes months to introduce a change). Second, these applications do not satisfy the flexibility and growth requirements of modern organizations. Third, many off-the-shelf C/S packages with nice GUI are becoming available. Finally, new employees don't want to work on legacy systems.

Dealing with such systems is very costly because of the complexity of understanding data semantics which are either buried in application programs or were never documented by original designer. The incompleteness of their specifications leads to ambiguities of the interpretation of the data schema. The hardest case is when data resides in files, but understanding unnormalized and poorly documented relational databases also is very difficult ([Hainaut, 1996], [Parent, 1998]).

1.2.2 Distribution

In many environments and applications, existing data are usually stored in multiple legacy databases, managed by different DMS. These databases may be stored on one or more computer systems that are either centrally located or geographically distributed.

1.2.3 Autonomy

Legacy database systems were typically designed to support local requirements imposed by a local environment, and without considering a possible cooperation with other systems. In other words, databases are usually under separate and independent control. The different aspects of autonomy are summarized as follows [Sheth, 1990]:

- *Design autonomy*. The databases have their own data model, query language, semantic interpretation of data, constraints, etc.
- *Communication autonomy*. The databases have the ability to decide when and how to respond to requests from other databases.
- *Execution autonomy*. The execution order of transaction is controlled by the legacy databases. They don't need to inform any other system of the execution order of local or external operations.
- *Association autonomy*. The legacy databases are able to decide whether participate or not in one or more federations, as well the possibility of its dissociation of a federation.

It is desirable to preserve the autonomy of the legacy databases. First, because a legacy database was originally an independent database system, it may have had many application programs developed on it. Such applications should continue to be executable in a legacy database. Second, legacy databases often belong to different organizations that maintain full control over their data. It is desirable for these organizations to retain a high degree of control of their legacy databases.

1.2.4 Heterogeneity

A major obstacle to interoperability of legacy databases is their heterogeneity. Heterogeneity among legacy databases is caused by the design autonomy of their owners in developing such systems. Legacy systems were typically designed to support local requirements, under constraints imposed with a given system. We can distinguish several types of heterogeneity [Thiran, 1998]: the platform, DMS, location and semantics level. The *platform level* copes with the fact that databases reside on different brands of hardware, under different operating systems, and interacting through various network protocols. Leveling these differences leads to platform independence. *DMS level* independence allows programmers to ignore the technical detail of data implementation in a definite family of models or among different data models. Representing data with different data models creates heterogeneity because of the inheriting expressive powers and limitations of DMS data models [Ozsu, 1991]. *Location independence* isolates the user from knowing where the data reside. Finally, *semantic level independence* solves the problem of multiple, replicated and conflicting representations of similar facts.

Current technologies such as de facto standards (e.g., ODBC and JDBC), or formal bodies proposals (e.g. CORBA, EJB), now ensure a high level of platform independence at a reasonable cost, so that this level can be ignored from now on. DMS level independence is effective for some families of DBMS (e.g. through ODBC or JDBC for RDB), but the general problem

is still unsolved when several DMS models, including legacy ones, are to cooperate. Location independence is addressed either by specific DBMS (e.g. distributed RDBMS) or through distributed object managers such as CORBA middleware products. Despite much effort spent by the scientific community, semantic independence still is an open and largely unsolved problem ([Aslan, 1999], [Härder, 1999]).

1.2.5 Mediation

To address the problem of interoperability of information systems in general, the term *mediation* has been defined in [Wiederhold, 1995] as a service that links data resources and application programs. A *mediator* is a software module that exploits encoded knowledge about some sets or subsets of data to create information for applications [Wiederhold, 1992]. Tasks involved in mediation include [Vermeer, 1996]: (1) accessing and retrieving relevant data from multiple heterogeneous sources, (2) transforming retrieved data to be integrated, (3) integrated the homogenized data, (4) managing the instance and structural conflicts, and (5) reducing the integrated data by abstraction. Several prototype mediator systems have been developed (e.g., [Garcia, 1995], [Vermeer, 1996]).

1.2.6 Mediation and Legacy Databases

A legacy database federation can be seen as a special case of mediation, where all data sources are legacy databases (i.e., heterogeneous and autonomous) and the mediator offers a virtual and integrated view of the underlying legacy databases.

A legacy database federation performs mediation by using a hierarchy of mediators that dynamically transform queries based on a federated schema into physical queries based on the physical schema of the legacy database sources (Cf. Figure 1).

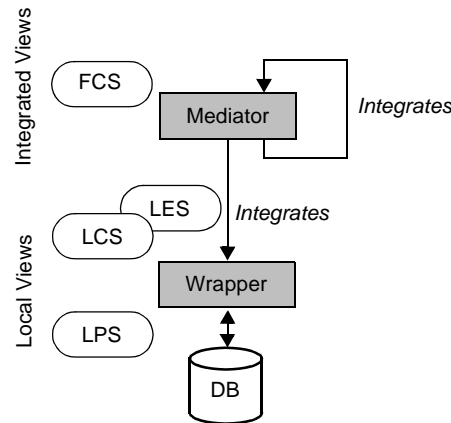


Figure 1-1: A general architecture of database federation.

Hierarchy architecture

The hierarchy architecture of a federation in general has been described in [Sheth, 1990]. It consists of a hierarchy of data descriptions that ensure independence according to different dimensions of heterogeneity. According to this framework and according to the legacy nature of the database source, each local database source is described by its own *physical schema* (LPS) from which a semantically rich description called *conceptual schema* (LCS), is obtained through a database reverse-engineering process. From this conceptual view, a subset called *export schema* (LES) is extracted. All the export schemas are merged into the *federated schema* (FCS). The federated schema as well as the conceptual and export schemas are expressed in a *canonical data model* (CDM) which is independent of the underlying technologies.

Component architecture

The function of a mediator is to provide integrated information, without the need to integrate the data resources. A mediator hides details about the location and representation of relevant data to applications.

On top of each legacy database is a *wrapper*. A wrapper is a software component that performs the translation between the export schema and the physical schema of the database [Papakonstantinou, 1995]. That is, the wrapper (1) offers an export schema in the canonical data model (2) accepts queries against the export schema and translates them into queries understandable by the underlying database, and (3) transforms the results of the local queries into a format understood by the application. Wrappers and mediators relies on schema descriptions and mappings to translate queries and to form the result instances.

Heterogeneity issues

The architecture model depicted in Figure 1-1 provides an adequate framework for solving the heterogeneity issues discussed above [Thiran, 1998]. DMS and local semantic independence is guaranteed by the wrappers. Location and global semantic independence is ensured by the mediators. It provides data federated access irrespective of their location and resolves semantic conflicts. Finally, platform independence is ensured by both the wrappers and ad hoc middleware such as commercial ORB.

1.2.7 Database Integration

The current methodologies developed for building a database federation are generally based on a database integration approach (e.g., [Batini, 1986], [Schmitt, 1996], [Parent, 1998], [Hainaut, 1999]).

Referring to [Parent, 1998], the database integration is made up of four main processes: (1) preparing the database schemas; (2) integrating them; (3) defining the mappings and (4) building the architecture components.

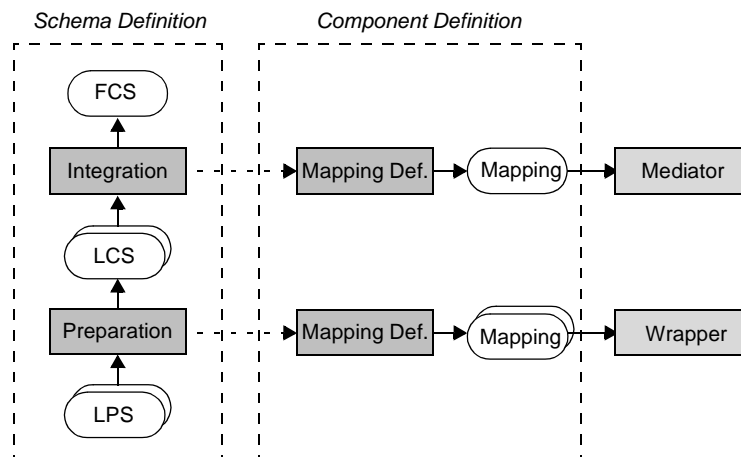


Figure 1-2: A general methodology for building a database federation.

Preparation

In this phase, schemas that correspond to the information sources being integrated are translated into schemas using the canonical data model. It includes two main tasks [Parent, 2000], namely, syntactic rewriting and semantic enrichment.

Syntactic rewriting. Local schemas are translated into a canonical data model. This allows for resolving syntactic heterogeneity that is the result of different data models.

Semantic enrichment. This is the process that aims at augmenting the knowledge about the semantics of data. Extracting a semantically rich description from a data source is the main goal of the data reverse engineering process (DBRE). Reverse engineering relies on the analysis of whatever information is available: schema specifications, index definitions, data, queries in existing programs. Combining these analysis makes it possible to recover hidden structures and constraints [Hainaut, 1996].

Integration

The integration is the process of identifying similar components in export schemas, identifying and solving the conflicts in these schemas, and finally, merging export schemas into a federated one.

Conflicts fall into three possible categories: syntactic, semantic and instance. Besides the usual conflicts related to synonyms and homonyms, a *syntactic conflict* occurs when the same concept is presented by different object types in local schemas. For instance, an *OrderDetail* can be represented by an entity, by an attribute value and by a relationship. A *semantic conflict* appears when a contradiction appears between two representations A and B of the same application domain concept or between two integrity constraints. Solving such conflicts uses reconciliation techniques, generally based on the identification of set-theoretic relationships between these representations: $A = B$, $A \text{ in } B$, $A \text{ and } B \text{ in } AB$, etc. It is based on set-theoretic relations existing among the instances of data types, and that may conflict with semantic reconciliation. Instance conflicts are specific to existing data. Though their schemas agree, the instances of the databases may conflict. As an example, common knowledge suggests that *USER* be a subtype of *EMPLOYEE*. However, data analysis shows that $\text{inst}(\text{EMPLOYEE}) \supsetneq \text{inst}(\text{USER})$, where $\text{inst}(A)$ denotes the set of instances of data type A. This problem has been discussed in [Vermeer, 1996]. This process is highly knowledge-based and cannot be performed automatically.

Solving the conflicts occurring in heterogeneous databases has been studied in numerous references, by e.g. [Spaccapietra, 1991], [Batini, 1986] and [Vermeer, 1996]. It is important to note that most conflicts can be solved through three main techniques that are used to rework the local schemas before their integration: renaming, transforming and discarding. Heuristics exist to cope with this problem [Spaccapietra, 1991].

Defining the mappings

An analysis of the schema recovery and integration processes shows that deriving a schema from another one is performed through techniques such as renaming, translating, solving conflicts, which basically are schema transformations. Most data-centered engineering processes can be formalized as a chain of schema transformations. This is the case for RDBE and integration [Hainaut, 1999].

Both queries translation and results building rely in mappings and schemas description. The mappings are pure transformational functions that cannot be immediately translated into ex-

ecutable procedures in 3 GL. However, it is fairly easy to produce procedural data conversion programs [Thiran, 1999].

1.3 Scope and Overview

1.3.1 Scope

The InterDB project proposes a general *architecture*, a *methodology* and a *CASE environment* intended to address the problem of providing users and programmers with an abstract integrated view to independent, heterogeneous and distributed databases.

1.3.2 Overview

We distinguish three main tasks addressing the InterDB scope: (1) defining a generic integration framework intended to express all the federation components, schemas and mappings; (2) building the wrappers for legacy source databases; and (3) defining a meta-mediator instantiated for federated schema.

Defining a generic integration framework

This issue is discussed in Part I of this report. It involves defining a unique and generic framework intended to express all the federation components, schemas and mappings.

In *Chapter 2*, we introduce the general architecture of database federations. We provide an overview of existing database federation architectures. We then introduce the main baselines of the InterDB architecture.

In *Chapter 3*, we present the generic data model intended to express the schema hierarchy of database federations. It is an abstract formalism from which the federation models can be derived by specialization. In short, physical schemas, conceptual schemas, export schemas as well as federated schemas are expressed into an unique and generic entity/object-relationship model. Besides the standard concepts, the meta-model includes some meta-objects which can be customized according to specific needs. These features provide dynamic extensibility of the generic model. For instance, new concepts such as correspondence types can be represented by specializing the meta-objects.

In *Chapter 4*, we define the mappings as schema transformations. We present the concepts and properties of schema transformations. An inventory of useful transformations is presented. We finally introduce the notion of schema history.

Building wrappers for legacy databases

This issue is discussed in Part II of this report. We present an architecture, a methodology and

a CASE support for developing *legacy data wrappers*.

In *Chapter 5*, we present and develop the technology of legacy data wrappers. We discuss their main roles and services they provide. In particular, we show the close link between reverse engineering and such wrappers. The architecture of an operational data wrapper - the InterDB wrapper - is then presented.

In *Chapter 6*, we present a generic and complete methodology for building legacy data wrappers. The methodology includes the schema recovery through a reverse engineering approach and the mapping building.

In *Chapter 7*, the methodology is supported by the DB-MAIN CASE tool that gives users an integrated toolset for reverse engineering and inter-schema mapping definition and processing. Wrapper generators for Cobol files and relational databases have been written as add-ons to DB-MAIN.

Defining the mediator

This issue is discussed in Part III of this report. We present an architecture, a methodology and a CASE support for developing *mediators*.

In *Chapter 8*, we present and develop the technology of data mediators. As for the wrappers, we start by discussing the main roles and services mediators provides. This chapter shows, among others, how queries against a federated schema can be processed in terms of queries against the underlying wrappers. Finally, the architecture of the InterDB mediator is presented.

In *Chapter 9*, we present an overview of the database integration methodology. The issues are raised and the approaches that have been proposed to tackle the problem are discussed. The InterDB approach is then presented and its main characteristics are outlined.

In *Chapter 10*, we present the DB-MAIN tools that are intended to support the mediator development. In particular, this chapter presents the extension of the DB-MAIN repository that describes both the schema hierarchy and the mappings between them.

Part I

Generic Integration Framework

Integration Architecture

In which an overview of classical integration architecture of databases is given. The InterDB architecture is then presented and its main characteristics are outlined.

2.1 Introduction

Database systems that provide interoperation and varying degrees of integration among distributed existing databases have been termed *multidatabase systems* ([Hurson, 1994], [Litwin, 1986], [Litwin, 1994]), *federated databases* ([Heimbigner, 1985], [Sheth, 1990]), and more generically, *Heterogeneous Distributed Database Systems* (HDDBS) [Bougettaya, 1998]. An attempt to relate some of the frequently used terms, using the fundamental dimensions of distribution, heterogeneity and autonomy has been presented in [Sheth, 1990].

The chapter is organized as follows. First, we present a taxonomy that classifies the existing solutions into three categories: global schema integration, federated databases and multidatabase language approach. Second, we describe the InterDB architecture

2.2 Overview of integration architectures

In [Sheth, 1990], a reference HDDBS architecture has been presented (Cfr. Figure 2-1). This architecture is based on mappings between schemas on 5 levels.

- *Physical schema (LPS)*. A physical schema represents data in a data source. There is one physical schema for each data source. The physical schemas are expressed using a local data definition language and a local data model, if such exist.
- *Component schema (LCS)*. A component schema is a CDM¹ representation of a local schema. The local schema is translated into a CDM representation if the CDM is different than the local data model, otherwise the local and the component schemas are the same.
- *Export schema (LES)*. In some architectures, each data source decides the portion of the data that are going to be available for non-local access. The export schema models the view of the component schema visible non-locally. It is also expressed in the CDM.
- *Federated or Global schema (FCS or GCS)*. A federated (global) schema is an integration of all export schemas. Depending on the particular framework applied, this schema can be called either global or federated. The term global schema is used when there is only one such schema. There can be more than one federated schema.
- *External schema*. An external schema represents a subset of the global schema tailored for a particular user or group of users.

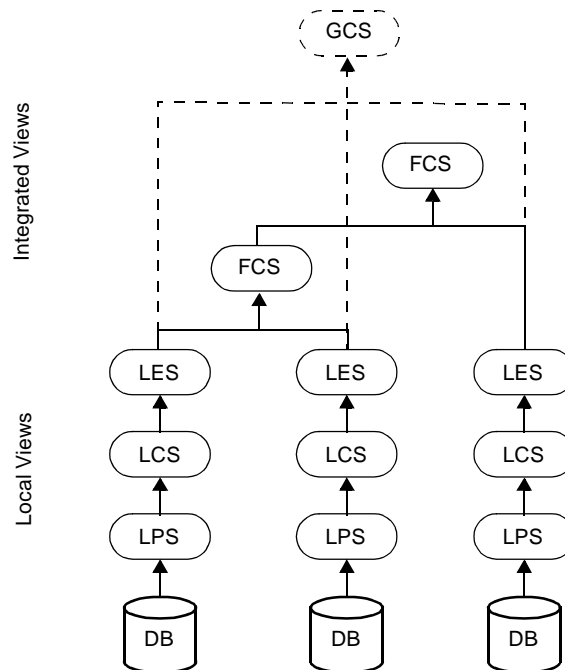


Figure 2-1: A Generic integration architecture. For simplicity, the export schemas have been ignored.

1. Canonical Data Model.

Depending on the level of integration, integration architectures can be classified into 3 categories: *global schema systems*, *multidatabase language systems* and *federated databases*. These categories reflect design efforts to accommodate the conflicting requirements of achieving an efficient and usable systems by larger level of sharing on one side, and preserving the autonomy of the data sources, on the other. On the one extreme of this spectrum are the systems that are closed to the distributed databases in building a global integrated schema of all the data in the sources. The opposite side represents systems that provide just basic interoperation capability and leave most of the integration problem to the user. The rest of this section overviews the features of each of these categories.

2.2.1 Global Schema Systems

A straightforward approach to building an HDDBS is the approach where the export schemas of multiple databases are integrated into a single view (global) schema [Spaccapietra, 1994]. In [Batini, 1986], a thorough survey on schema integration is provided and twelve methodologies are compared.

The user is not aware of the distribution and the heterogeneity of the integrated data sources. Multiple databases logically appear as one single database to users. Furthermore, if the schema does not change frequently, it can be stored locally, at the client, for faster access. Nevertheless, this approach has been shown to exhibit the following problems [Bouguettaya, 1998]:

- Since the general problem of integrating even only two schema is undecidable, the process of integration of multiple schemas is very hard to automate. Global schema integrators must be familiar with all the naming and structure conventions of all the data sources and integrate them into a cohesive single schema without changing the local schemas.
- There are two basic approaches to integrating the component schemas into a global schema. In the first, the component schemas are integrated pair-wise. A hierarchical application of the integration leads to a schema integrating all component schemas. The other approach is to integrate all the component schemas at once. Both approaches have problems. The first one could produce different results when different integration orders are used, while the other one is usually too difficult.

The global schema integration is not suitable for frequent dynamic changes of schemas as the whole process of integration may need to be redone. As a result, it doesn't scale well with the size of the database networks.

2.2.2 Multidatabase Languages

This approach does not provide any type of global schema. The only means of accessing the data in the data sources is by language primitives for specification of queries over data stored in multiple sources. Information stored in different sources may be redundant, heterogeneous and inconsistent. These problems occur when component system are strongly autonomous.

The aim of a multidatabase language is to provide accesses involving several databases at the same time. Such language has features that are not supported in traditional languages. For instance, a global name can be used to identify a collection of databases. Queries can specify data from any local participating database (example: MSQL [Litwin, 1994]).

The main criticism of the multidatabase language approach is the low level of transparency provided to the user. The user is responsible for finding the relevant information, understanding each database schema, detecting and resolving the semantic conflicts, and finally, building the required view of the data in the sources. The advantages of the approach are that it is not intrusive against the autonomy of the data sources and there is no global/federated schema maintenance and access overhead.

2.2.3 Federated Architecture

In the federated MDBMS (FDBS), the export schema are only a subset of the component schemas. The federated schema does not need to be an integration of all the export schemas. It can be integrated only portions of the export schemas of interest to the users using the federated schema. More than one federated schema can be defined according to users requirements. Each user can then further refine its export schema to fit his own requirements.

The aim of this architecture is to remove the need for static schema integration. It allows each local database to have more control over its sharable information. It should be noted that FDBS is a compromise between no integration and total integration. A typical FDBS architecture would have a common data model and an internal command language.

The level of integration and services in a FDBS depends on how tightly/loosely coupled the component DMS are.

Tightly coupled systems

In a tightly coupled FDBS, federation administrators have full control on the creation and maintenance of federated schemas and access to export schemas. The aim is to provide location, replication and distribution transparency. This approach supports one or more federated schemas.

A federation repository keeps the mappings between the different schemas and helps maintain uniformity in the semantic interpretation of multiple integrated components of data.

The size of the repository can grow dramatically as the number of data sources and users increase. It can also become a performance bottleneck when accessed by a large number of users. These problems are reminiscent of the problems of maintaining a global schema described above.

Once a federated schema is created, it is rarely changed; that is, it is static. This, it does not support dynamic changes of export/component schemas.

Loosely coupled systems

Loosely coupled systems do not have a centralized administrator. The user creates and maintains his own integrated schema in the form of a local view. Creating a federated schema corresponds to creating a view against the relevant export schemas. In that respect, each user must be knowledgeable about the information and structure of the relevant export schemas in order to create views. Federated schemas here are dynamic and, as a result, can be created and dropped on the fly. Multiple federation schemas are supported. The maintenance problems noted above disappear.

A possible drawback of this approach is that more than one user might need to perform the same view modeling, without the possibility of reusing the definitions. Furthermore, a change in an export schema affects all the users who have a view dependent on it.

A solution to the problems noted above is to allow a gradual transition from the federated into export schemas by a hierarchy of small intermediate schemas. This approach breaks the repository into smaller and more maintainable units, while allowing reuse of the view specification and modularity in the view definition and change.

Mediation architecture

To address the problem of interoperability of information systems in general, the term mediation has been defined in this context [Wiederhold, 1995] as a service that links data resources and application programs. The function of a mediator is to provide integrated information, without the need to integrate the data resources. A mediator hides details about the location and representation of relevant data to applications. Several prototype mediator systems have been developed ([Wiederhold, 1992], [Tomasic, 1996], [Garcia, 1997]).

Figure 2-2 shows the basic architecture of information processing using mediators. Compared to the client/server architecture, mediation introduces an additional layer in the architecture of information systems.

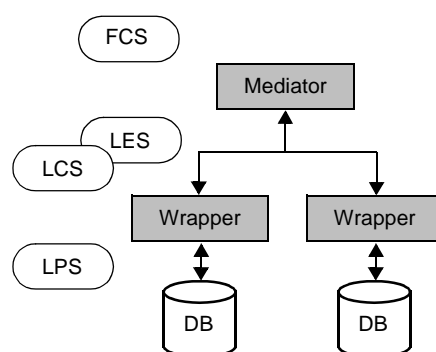


Figure 2-2: Wrappers/mediators architecture.

Mediation and databases

A database federation offers a virtual, integrated view of the underlying component databases. Queries issued against this view are translated into queries against the underlying component databases.

A legacy database federation performs mediation using at least two important components [Vermeer, 1997]: wrappers and mediators.

The function of a *mediator* is to provide integrated information, without the need to integrate the data resources. A mediator hides details about the location and representation of relevant data to applications.

Above each legacy database is a *wrapper*. A wrapper is a software component that performs the translation between the export schema and the physical schema of a source [Papakonstantinou, 1995]. That is, the wrapper (1) offers an export schema in the canonical data model (2) accepts queries against the export schema and translates them into queries understandable by the underlying database, and (3) transforms the results of the local queries into a format understood by the application.

Wrappers and mediators relies on schemas descriptions and mappings to translate queries and to form the result instances.

2.3 InterDB Architecture

InterDB is a mediator/wrapper system for integrating pre-existing heterogeneous, distributed and legacy databases. It provides users with an unified federated schema and a single, high-level object-oriented interface to access data from any of the integrated local databases. Local databases remain fully autonomous.

2.3.1 Hierarchy Architecture

The InterDB hierarchy architecture is based on the five levels federated architecture [Sheth, 1990] with some simplifications (Figure 2-3). It comprises four schema levels:

- *Local Physical Schema (LPS)*. It describes the physical data structure of the database as they are implemented by the data manager. It holds structures and constraints explicitly declared in the DDL schema declaration or in data dictionaries.
- *Wrapper Logical Schema (WLS)*. It is the description of the data structures perceived by users and programmers. In other words, it is the legacy database view offered by the wrapper. In the InterDB approach, the wrapper logical schema also includes implicit constraints, that is, data properties that have not been explicitly declared, but that are managed by, say, application programs.

- *Wrapper Object-oriented Schema (WOS)*. It is the object-oriented definition of a logical schema. A wrapper logical schema is translated into an object-oriented schema through one-to-one mappings that convert entity types into object types and attributes into methods.
- *Federated Object-oriented Schema (FOS)*. It is an object-oriented federated view that integrates the local objects of WOS. WOS and FOS are represented in the same object-oriented formalism.

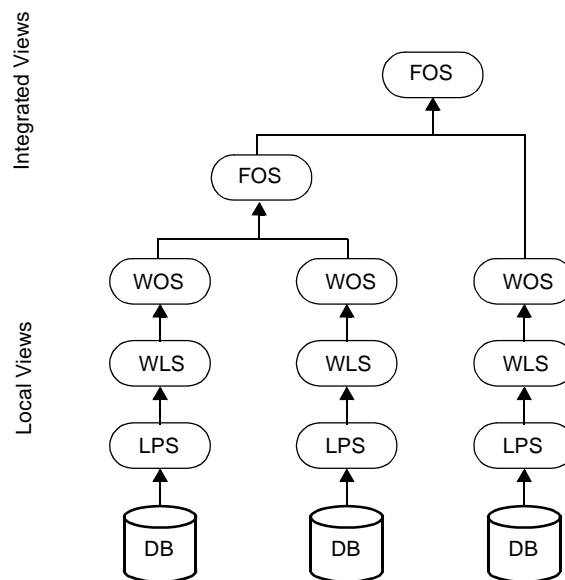


Figure 2-3: InterDB hierarchy architecture: local physical schemas, wrapper logical schemas, wrapper object schemas and federated object schemas.

2.3.2 Component Architecture

The InterDB architecture, shown in Figure 2-4, is close to standard proposals. It comprises a hierarchy of mediators, namely the legacy data wrapper dedicated to each database and a mediator based on the federated object schema (FOS). These mediators offer remote Java objects that hide the data distribution and the federation heterogeneity.

We have used the RMI system to manage the communications between the wrappers and the mediators.

Wrapper. A *wrapper* is in charge of managing the physical/object conversion of each local database. It comprises two components, namely the logical wrapper and the object wrapper. The *logical wrapper* hides the syntactic idiosyncrasies and the technical details of the DMS

of a given model family. In addition, it makes the implicit constructs and constraints explicitly available. For instance, relational databases and flat COBOL files appear as similar logical structures. A logical wrapper dynamically transforms queries (top-down) and data (bottom-up) from this logical model to the actual physical model. In particular, it emulates implicit constructs such as foreign keys in COBOL files or multivalued fields in relational DB. The *object wrapper* provides a remote object-oriented view of a local logical database. It appears as a remote object server that offers a unique abstract interface to Java programs. For performance reason, we have decided to develop the wrappers as program components dedicated to a local database. In particular, the logical/physical and object/logical mapping rules are hardcoded in the modules rather than interpreted from mapping tables.

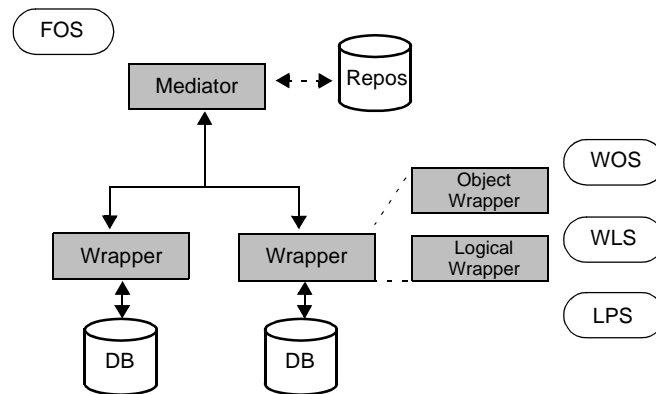


Figure 2-4: The InterDB component architecture of a federated database. To simplify the figure, export and view schemas have been ignored.

Mediator. The *mediator* offers an object-oriented interface based on the FOS. It provides global remote objects that hide data distribution across the different sites. For flexibility reason, the mediator is based on the DB-MAIN repository that describes the federated object-oriented schema, the local object-oriented schema of each wrapper, its location, and the relationships between local and global schemas. Information concerning data replication, semantic conflicts and data heterogeneity allows the server to interpret and distribute the global objects, and to collect and integrate the results sent back by the local wrappers.

The architecture model depicted in Figure 2-4 provides an adequate framework for solving the heterogeneity issues discussed in Chapter 1. DMS and local semantic independence is guaranteed by the local wrappers. Location and global semantic independence is provided by the mediator. This module provides data global access irrespective of their location and resolves semantic conflicts. Finally, platform independence is ensured by both the local wrappers, Java and RMI as a middleware.

Generic Data Model

In which the generic data model of the federation is presented. The generic model is able to describe data structures at different levels of abstraction, ranging from physical to conceptual, and according to various modeling paradigms.

3.1 Introduction

Over the years, several data modelings have been used as legacy data models: hierarchical, network, relational, semantic and object-oriented models. Schemas that correspond to these legacy data models are translated into schemas using a *Canonical Data Model* (CDM). This allows for resolving syntactic heterogeneity that is the result of different data models. For example, in the Multibase system, the legacy DMS are relational and network systems, and the CDM follows the functional model.

It is usually expected that the modeling power of the CDM is richer than the legacy data models. The *relational model* has frequently been used as the CDM with relational, hierarchical and network databases. Since the *entity-relationship model* has been the overwhelming tool for conceptual modeling, early efforts in data modeling translation research focused on the transformation to and from the ER model. Next, there has been a shift to using the *object-oriented model* as the focal model through which other models have to be translated to or from ([Urban, 1991], [Vermeer, 1996], [Roantree, 2001]). The shift has been spurred on by the fact that the object-oriented model can be used as a tool for both design and implementation. The current tendency is to use *XML* as the CDM ([Manolescu, 2001], [Gardarin, 2002]). This is advocated for interoperable systems because of the ease of representing both structured and semi-structured data. Another reason for choosing XML as a standard for informa-

tion interchange is its flexibility, portability and simplicity [Manolescu, 2001]. An interesting discussion on the different models used as CDM can be found in [Elmagarmid, 1999].

In the InterDB approach, we define a high-level generic data model, namely the *generic data model*, such that it is possible to represent the constructs whatever their underlying data model and their abstraction level. As we will see in the next sections, the generic data model can be used as a unifying model for any legacy and canonical data models.

3.2 Generic Data Model

The *generic data model* is an abstract formalism intended to express data structures independently of the implementation technologies. For methodological reason, we propose a unique generic model from which several abstract submodels can be derived by specialization. In short, physical schemas, wrapper logical schemas as well as object-oriented schemas are expressed into an unique and generic entity/object-relationship model (Figure 3-1).

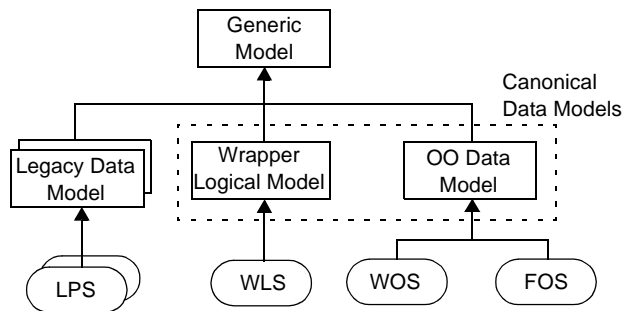


Figure 3-1: The data model hierarchy.

Besides the standard concepts, the generic data model includes some meta-objects which can be customized according to specific needs. These features provide dynamic extensibility of the generic model. For instance, new concepts such as correspondence types can be represented by specializing the meta-objects.

3.2.1 Main concepts

The main concepts of the generic model are illustrated graphically in Figure 3-2. Figure 3-3 summarizes the six major constructs of the generic data model. The central construct is that of entity type, or object type (Customer), that represents any homogeneous class of conceptual, component or physical entities, according to the abstraction level at which these entities are perceived. Entity types can have attributes (CustCode, Price, UnitPrice), which can be

atomic (QtyOH) or compound (Price), single-valued (Name) or multivalued (Price), mandatory (Name) or optional (Phone). Cardinality [i-j] of an attribute specifies how many values (from i to j) of this attribute must be associated with each parent instance (entity or compound value). The values of some attributes, called reference attributes (DETAIL.ItemCode), can be used to denote other entities (i.e., they form some kind of foreign keys). Relationship types (places, has) can be drawn between entity types. Each of their roles (places.ORDER) is characterized by a cardinality constraint [i-j], stating that each entity must appear in i to j relationships. Additional constraints such as identifiers made of attributes and/or roles as well as existence constraints (coexistence, exclusive, at-least-one, etc.) can be defined. Constructs such as access keys (ITEM.{Name}), which are abstractions of such structures as indexes and access paths, and storage spaces (File_DOC) which are abstractions of files and any other kinds of record repositories, are components of the generic model as well. A processing unit (CUSTOMER.Remove) is the abstraction of a program, a procedure or a method, and can be attached to an entity type, a relationship type or a schema.

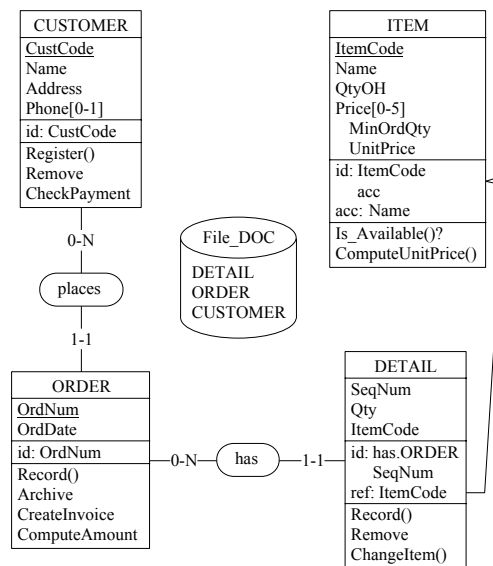


Figure 3-2: An illustration of the generic model. This schema includes entity types, relationship types, attributes, identifiers and processing units. It also includes foreign keys, access keys and storage spaces. This hybrid schema includes constructs from different levels of abstraction.

Entity/object type	Category of similar data/information units
Attribute	Common property of the entities of a given type; atomic/compound, single-valued/multivalued, optional/mandatory, value/entity-based
Relationship type	Type of aggregate comprising roles and attributes
Group	List of attributes/roles attached to a parent (entity type, rel-type, compound attribute); can be given functions: identifier, existence constraint, access key, etc.
Inter-group relationship	dependency between groups; example: foreign key, functional dependency, inclusion constraint
Collection	set of entities

Figure 3-3: The six main concepts of the generic data model.

3.2.2 Meta concepts

Besides the main concepts, the generic data model includes some *meta objects* which can be customized according to specific needs.

- *Meta-property*: user-defined property of a construct, in addition to built-in properties such as name, type, length, semantic, etc.
- *Stereotype*: a specific category of a construct; an instance of a construct can belong to zero, one or several stereotypes; a stereotype can be given specific properties and specific behaviour.

These features provide dynamic extensibility of the generic model. For instance, new concepts such as organizational units, servers, or geographic sites can be represented by meta-properties. Following [Busse, 1997], we can distinguish the following kinds of meta-properties:

- *Mapping meta-property* describes the correspondence between constructs of two different schemas.
- *Technical meta-property* describes information regarding the technical access mechanisms of components, such as the protocol, speed of connection, cost of queries, query capabilities and so on. It is used to bridge technical and interface heterogeneity.
- *Semantic meta-property* is information that helps to describe the semantic of concepts. In particular, ontologies and thesauri are used for this purpose. All domain-specific descriptions belong to this class.
- *Quality-related meta-property* describes source-specific properties of information systems regarding their quality, such as reliability, update frequency, actuality, comprehensiveness, etc. This is used for ranking or optimization.

- *User-related meta-property* describes responsibilities and preferences of users of the information systems, e.g. user profiles.

3.2.3 Model specialization

This generic model can be specialized into the legacy data model, the wrapper data model and the canonical data model. These models are built by selecting generic constructs and structural constraints, and by renaming constructs to make them comply with the concept taxonomy of the specialized model. Figure 3-4 shows some common interpretation of the generic constructs. For example, the relational model, considered as a legacy data model, can be precisely defined as follows (IMS, Cobol or OO models can be defined in the same way):

- *Selecting constructs.* We select the following constructs: entity types, attributes, identifiers and reference attributes.
- *Structural constraints.* An entity type has at least one attribute. The valid attribute cardinalities are [0-1] and [1-1]. An attribute must be atomic.
- *Renaming constructs.* An entity type is called a table, an attribute is called a column, an identifier, a key and a group of reference attributes, a foreign key.

In the same way, an object-oriented data model (e.g., a variant of the UML class model) can be described as follows:

- *Selecting constructs.* We select the following constructs: entity types, IS-A relations, processing units, attributes, relationship types, identifiers.
- *Structural constraints.* An entity type has at least one attribute. A relationship type has 2 roles. An attribute is atomic. The valid attribute cardinalities are [0-1] and [1-1]. An identifier is made up of attributes, or of one role + one or more attributes. Processing units are attached to entity types only.
- *Renaming constructs.* An entity type is called a class, a relationship type is called an association, a processing unit is called an operation, an attribute is an attribute, the cardinality of the opposite role is called multiplicity and an identifier comprising a role is called a qualified association.

Generic	ER	Relational	COBOL
Entity/object type	Entity type	Table	Record type
Attribute	Attribute	Column	Field
Relationship type	Relationship type		
Group	Identifier Constraint	Primary key Foreign key Index	Record key
Collection		Table space	File

Figure 3-4: Some common interpretations of the generic concepts.

3.3 Federation Data Models

The federation data models include the legacy data models supported by the legacy databases and the canonical data models. In this section, we present these models and illustrate them by a small common example. These models are interpreted as specializations of the generic model described above.

3.3.1 Legacy Data Models

Over the years, several data modelings have been used to design universes of discourse: relational model, network model (CODASYL DBTG), hierarchical model (IMS), shallow model (TOTAL, IMAGE), inverted file model (DATACOM/DB), standard file model (COBOL, C, RPG, BASIC) or object-oriented model.

Due to the large variety of model families, it is not easy to propose an exhaustive description of their own constructs and constraints. As far as the InterDB project is concerned, we will consider two popular legacy data models only: the COBOL model and the relational model.

COBOL data model

The COBOL data model imposes few constraints on attribute structures (Figure 3-5). The most important one concerns multivalued attributes, which can be represented through list attributes only. In addition, optional attributes are not explicitly represented except as multivalued attributes.

An example of a COBOL schema is shown in Figure 3-6. In this model, record types have

only one record key (e.g. Customer has one attribute that plays the role of identifier and access key). Attributes are atomic or compound (e.g. address) or multivalued (e.g. phone). All the attributes are mandatory. Names are formed according to the COBOL language syntax.

Generic Model	COBOL Model	Constraint
Entity/object type	Record Type	
Attribute	Field	Mandatory
Single-value atomic attribute	Single-value elementary field	Mandatory
Compound attribute	Compound field	Mandatory
Multivalued attribute	<i>... occurs N times</i>	Mandatory
Identifier + access key	Record key, alternate record key	Only one attribute
Non-identifier access key	Alternate record key <i>with duplicates</i>	Only one attribute
Collection	Files	

Figure 3-5: Concepts and constraints of the COBOL data model.

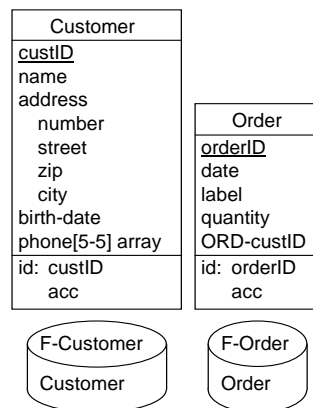


Figure 3-6: COBOL structure example.

Relational data model

The attribute structure in the relational data model is particularly poor (Figure 3-7): an attribute must be single-valued and atomic. It can nevertheless be optional. Moreover, the relational data model introduces the concept of foreign key that is not defined in the COBOL data

model.

Generic Model	Relational Model	Constraint
Entity/object type	Table	
Attribute	Column	Single-valued, atomic
Optional attribute	<i>nullable</i> column	Single-valued, atomic
Primary identifier	Primary key	Must be an index
Secondary identifier	unique (column/table predicate) unique index	Must be an index
Referential key	Foreign key	
Access key	Index	
Collection	Tablespace, DBspace, etc.	

Figure 3-7: Concepts and constraints of the relational data model.

An example of a relational data schema is shown in Figure 3-7. In this model, tables have primary and unique keys (e.g. Customer has two identifiers: primary and secondary). All the attributes are atomic and single-valued. Attributes can be mandatory or optional (e.g. phone). Tables can have one or several foreign keys (e.g. custID is a referential attribute of Order. It references custID attribute of Customer). Names are formed according to the relational language syntax.

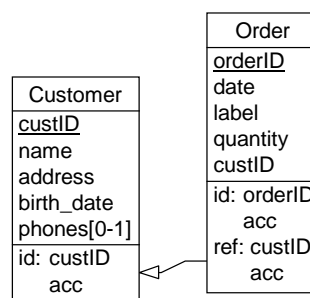


Figure 3-8: Relational schema example.

3.3.2 Canonical Data Models

A canonical data model is designed to express all the semantics of the local schemas [Sheth, 1990]. As a result, it is usually expected that its modeling power is richer than the data models

followed by the legacy databases. A canonical data model must therefore include at least all the structures and constraints of any underlying legacy data schemas based on the legacy data models.

In InterDB, we propose two canonical data models: the *wrapper logical model* associated with a common data manipulation language close to SQL and the *object-oriented model* that offers object methods for accessing read-only data.

Wrapper logical model

The *Wrapper Logical Model* (WLM) hides the syntactic idiosyncrasies and the technical details of the DMS of a given model family. Since we only consider the COBOL and relational data models as legacy data models, we defined WLM as a model that includes all the structures and constraints that exist explicitly in these two data models (Figure 3-9). As a result, WLM comprises entity types and attributes (that can be mono- or multi-valuated; atomic or compound, mandatory or optional) as constructs; identifiers and referential attributes as constraints.

An example of a schema of this model is shown in Figure 3-10. In this model, entity types may have one or two identifiers constituted of one or more attributes or roles (e.g. Customer has one identifier constituted of one attribute). Attributes are atomic or compound (e.g. address). Attributes may be mandatory (e.g. name attribute of Client) or optional (e.g. birth-date attribute of Customer). Attributes may also have several values (e.g. phone attribute of Customer). Entity types may have one or several referential attributes (e.g. custID is a referential attribute of Order. It references custID attribute of Customer). Names are formed according to host language syntax.

Constructs	Constraints
ET	Attributes: any number Identifier: any number
Attribute	Atomic / compound card: [1-1],[0-1], [0-i]
Attribute Domain	Char(n), Num(n), Num(n,m)
Identifier	n level-1 attributes
Referential attributes	n level-1 attributes
Names	Host language compliant

Figure 3-9: The wrapper logical model: constructs and constraints.

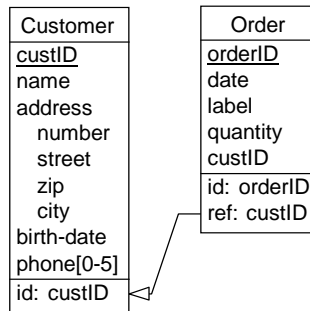


Figure 3-10: Wrapper logical schema example.

3.3.3 Object-oriented Model

The object-oriented model (Figure 3-11) provides rich data structuring possibilities, which enables them to express all the semantics of the wrapper logical schemas. Moreover, it permits the specification of behaviours (through the processing units), which can be used to perform complex mappings among the schemas of a database federation. Thus, we can use this model for modeling both the local and global schemas.

An *object type* (named entity type in the generic model) definition is structured in three sections: the *structure section*, in which for instance, attributes can be defined, a *constraint section* in which *constraint groups* are defined, and a method section in which operations on objects are defined.

Constructs	Constraints
ET	Attributes: any number Identifier: any number
Attribute	Atomic / compound card: [1-1],[0-1], [0-i]
Attribute Domain	Char(n), Num(n), Num(n,m)
Identifier	n level-1 attributes
Relationship type	one-to-one, one-to-many
Processing unit (method)	
Names	Host language compliant

Figure 3-11: The object-oriented data model: constructs and constraints.

Below we present a brief overview of how OO modeling constructs can be derived from a wrapper logical schema, with an illustration from our previous example schema (Figure 3-12).

Wrapper object-oriented schema

Object type are primary constructed from entity types of the wrapper logical schema. Such objects are called *entity objects*. The entity objects properties correspond to the attributes of the entity types. In the example schema of Figure 3-10, the entity types Customer, Order give rise to corresponding objects in the object schema.

Single-valued attributes are modeled as simple properties (e.g. integer, string or date) whereas multivalued attributes are modeled as vectors. In our example, the attributes custId and name give rise to corresponding Java objects (respectively Integer and String); the phone attribute is modeled as a Vector object.

In some cases, it is also possible to detect so-called *implicit entity objects*. These are entity object that have not been implemented by a entity type in the logical schema due to logical considerations. The logical schema characteristic leading to the discovery of a missing entity object is the existence of a complex attribute - i.e., (multivalued) compound attribute. This is the case for the address attribute in our example. It suggests an entity object which has four properties: number, street, zip, city and the corresponding properties.

Built-in properties are defined on attributes. A property returns the current instance of an attribute. A `getNumber` property is defined, for instance, on the Number attribute defined on the address entity object.

Relationship types are defined between (implicit) entity objects. Many-to-one or one-to-one relationships are supported. Relationship types connect either two entity objects that reference themselves or an implicit entity object to its source entity object. The property defined on a one-to-one relationship is `getEntityObjectName` whereas the properties defined on a many-to-one relationship are `getFirstEntityObjectName` and `getNextEntityObjectName`.

The translation between the wrapper logical and the object-oriented models will be discussed more in depth when we discuss the mapping definition (Chapter 4).

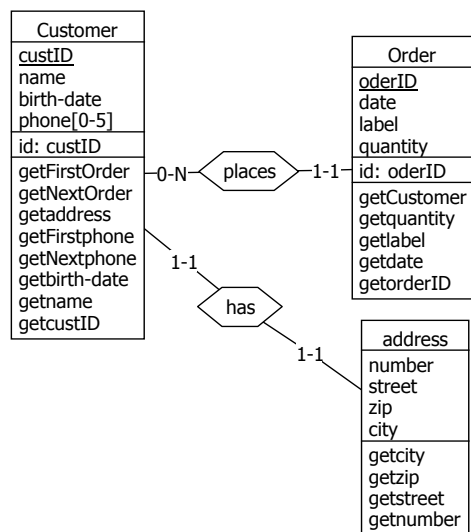


Figure 3-12: Wrapper object-oriented schema example.

Mapping Definition

In which the mappings are defined as schema transformation operators. The concept and some properties of schema transformations are presented and the notion of history is introduced.

4.1 Introduction

It can be shown that mappings can be modeled as data structure transformations. Indeed, the production of a schema can be considered as the derivation of this schema from a (possibly empty) source schema through a chain of elementary operations called schema transformations. Adding a relationship type, deleting an identifier, translating names or replacing an attribute with an equivalent entity type, all are examples of basic operators through which one can carry out such engineering processes as DBMS schema translation [Hainaut, 1993b; Rosenthal, 1988; Rosenthal, 1994], schema integration [Batini, 1992] or data conversion [Navathe, 1980]. As it will be shown later on, they can be used for integration engineering of legacy databases as well.

4.2 Mapping Baselines

Current mapping definitions of wrappers and mediators, such as TSIMMIS [Chawathe, 1994], InterViso [Templeton, 1995], IM [Levy, 1996] and Garlic [Roth, 1997], are what may be termed *query-oriented*. They provide mechanisms by which users define global schema constructs as view over source schema constructs (or vice versa in the case of IM), but do not

focus on the semantics of the data sources. More recent work on automatic wrapper generation ([Vidal, 1998], [Hammer, 1997]) and agent-based mediation [Bayardo, 1997] is also query-oriented.

In contrast, the InterDB approach is *schema-oriented* in that we provide mechanisms by which mappings are defined as schema transformations. These transformations are used to automate the translation of queries between the schema hierarchy.

This approach has several advantages over the query-oriented one [McBrien, 2000]:

- Focusing the human input to the integration process where it is most needed, namely on the semantics of the data sources, rather than on the more automatable query processing aspects;
- Decomposing the transformation/integration of schemas into a sequence of small steps by the provision of a set of primitive transformations which can be incrementally composed into more complex ones;
- Using the transformation pathways between schemas to automatically translate queries posed on a global schema to queries posed on a set of source schemas;
- Enabling the systematic repair of global schemas and global query translation in the face of evolving source schemas;
- Favorable framework for software production automation. Indeed, the transformation techniques can be completely formalized, and therefore translated into restructuration algorithms that can be the kernel of a generic CASE tool.

The generic model defined in Chapter 3 is the ideal support for schema transformation. Indeed, transformations can be used whatever their underlying data model and their abstraction level. For instance, the same schema transformation can be used in a relational model and in a conceptual one. The schema transformation definition on the generic model brings several important benefits:

- A transformation can be carried out for a construct of a modeling language M_1 where the result of this transformation is defined in terms of another language M_2 . This allows inter-model transformations to be applied, where the constructs of one modeling language are replaced with those of another.
- Such inter-model transformations form the basis for automatic inter-model translation of data and queries. This allows data and queries to be translated between different schemas in interoperating database architectures such as database federations and mediators.

4.3 Definition

A (schema) transformation is most generally considered as an operator by which a source data structure C is replaced with a target structure C' . Though a general discussion of the concept

of schema transformation would include techniques through which new specifications are inserted (*semantics-augmenting*) into the schema or through which existing specifications are removed from the schema (*semantics-reducing*), we will mainly concentrate on techniques that preserve the specifications (*semantics-preserving*).

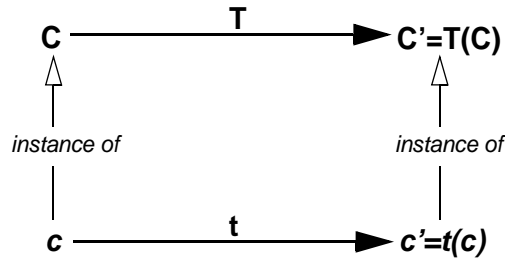


Figure 4-1: The two mappings of schema transformation $\Sigma \equiv \langle T, t \rangle$.

A transformation Σ can be completely defined by a pair of mappings $\langle T, t \rangle$ where T is called the *structural mapping* and t the *instance mapping*. T explains how to replace C with C' , while t states how instances of C must be replaced with instances of C' (Figure 4-1).

Another equivalent way to describe mapping T consists of a pair of predicates $\langle P, Q \rangle$, where P is the weakest precondition C must satisfy for T being applicable, and Q is the strongest postcondition specifying the properties of C' . So, we can also write $\Sigma \equiv \langle P, Q, t \rangle$.

4.3.1 Reversibility

Each transformation $\Sigma_1 \equiv \langle T_1, t_1 \rangle$ can be given an inverse transformation $\Sigma_2 \equiv \langle T_2, t_2 \rangle$, denoted Σ_1^{-1} as usual, such that, for any structure C ,

$$P_1(C) \Rightarrow C = T_2(T_1(C))$$

Σ_1 is said to be a *reversible transformation* if the following property holds, for any construct C and any instance c of C ,

$$P_1(C) \Rightarrow (C = T_2(T_1(C))) \wedge (c = t_2(t_1(c)))$$

So far, Σ_2 being the inverse of Σ_1 does not imply that Σ_1 is the inverse of Σ_2 . Moreover, Σ_2 is not necessarily reversible. These properties can be guaranteed only for a special variety of transformations, called *symmetrically reversible*.

Σ_1 is said to be a *symmetrically reversible transformation*, or more simply *semantics-preserving*, if it is reversible and if its inverse is reversible too. Or, more formally, if both following properties hold, for any construct C and any instance c of C ,

$$P_1(C) \Rightarrow (C = T_2(T_1(C))) \wedge (c = t_2(t_1(c)))$$

$$P_2(C) \Rightarrow (C = T_1(T_2(C))) \wedge (c = t_1(t_2(c)))$$

In this case, $P_2 = Q_1$ and $Q_2 = P_1$. A pair of symmetrically reversible transformations is completely defined by the 4-uple $\langle P_1, Q_1, t_1, t_2 \rangle$. Except when explicitly stated otherwise, all the transformations we will use in this presentation are semantics-preserving. In addition, we will consider the structural part of the transformations only.

We have discussed the concept of reversibility in a context in which some kind of instance equivalence is preserved. However, the notion of inverse transformation is more general. Any transformation, be it semantics-preserving or not, can be given an inverse. For instance, `del-ET(CUSTOMER)`, which removes entity type CUSTOMER from its schema, clearly is not a semantics-preserving operation, since its mapping t has no inverse. However, it has an inverse transformation, namely `create-ET(CUSTOMER)`. Since only the T part is defined, this partial inverse is called a structural inverse transformation. We will discuss these operators in more detail in the next sections.

4.3.2 Structural Analysis of a Transformation

The effect of a transformation T in schema S can be precised as follows. We define a schema S as a set of constructs C . Therefore, set-theoretic relations and operators apply on schemas. For instance, a schema can be declared as a subset to another one or can be defined as the union of the other schemas.

Let us consider the structural functions C_- , C_+ and C_0 :

- $C_-(T)$ returns the object of S that have disappeared in S' ;
- $C_+(T)$ returns the new object that appears in S' ;
- $C_0(T)$ returns the objects of S that are concerned by T , but that are preserved from S to S' .

4.3.3 Signature of a Transformation

A transformation can be specified through its *signature*, that states the name of the transformation, the names of the concerned objects in the source schema, and the names of the new objects in the target schema. For example, the signature of the transformations $T1$ and $T2$ in Figure 4-2 are as follows:

$T1: (R', \{(A, R1), (B, R2)\}) \leftarrow RT\text{-to-ET}(R)$

$T2: R \leftarrow ET\text{-to-RT}(R')$

The instance part of these transformations can be expressed as follow:

t1: for each $r(a,b)$ in R do:
 generate arbitrary entity r' in R'
 insert (r',a) in $R1$
 insert (r',b) in $R2$
 t2: $R = R1 \circ R2$ (where \circ means composition)

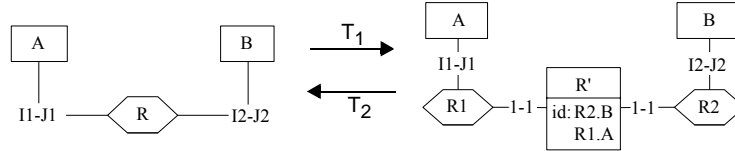


Figure 4-2: Representation of structural mapping T1 (from left to right) and T2 (from right to left) of a typical SR-transformations.

The first one is interpreted as "when applying RT-to-ET to relationship type R, the new entity type is called R', the rel-type involving A is called R1 and that involving B is called R2". The second one must read as follows: "when applying ET-to-RT to entity type R', the new rel-type is called R". The objects which are involved in the operation, but that can be identified in the schema from the names mentioned in the signature, are not specified. In the signature of T2 for instance, entity types A and B are not mentioned since they can be deduced as "all the entity types linked to R' in the source schema". A signature alone does not comprise the C-, C₊ and C₀ structural components, but it can be used to identify them in the source and target schemas. In addition, the format of a signature is not unique, but depends, a.o., on the default naming conventions. For instance, the roles are given default names in transformations T₁ and T₂ described above.

Just like transformations, signatures can be generic or instantiated. For instance, the generic signature

$$(R', \{(A, R1), (B, R2)\}) \leftarrow \text{RT-to-ET}(R)$$

could be instantiated, in an actual schema, into

$$(\text{ORDER}, \{(\text{CUSTOMER}, \text{from}), (\text{PRODUCT}, \text{of})\}) \leftarrow \text{RT-to-ET}(\text{order})$$

From these examples, we can observe an essential property of the signatures: their *reversibility*. Being provided with the right-side schema and the signature of T₂, we can derive the signature of T₁, and conversely. In other words, the signature provides enough information, not only for *redoing* the operation, but also to *undo* it.

4.3.4 Schema Transformation Sequence

A transformation sequence $T_p = \langle T_2 \circ T_1 \rangle$ is obtained by applying T2 on the schema that results from the application of T1. As an illustration, Figure 4-3 shows a sequence of two transformations usually used in database engineering process. The first one (T1) replaces a foreign key with a relationship type and the second one (T2) expresses a multiple attribute as an external entity type.

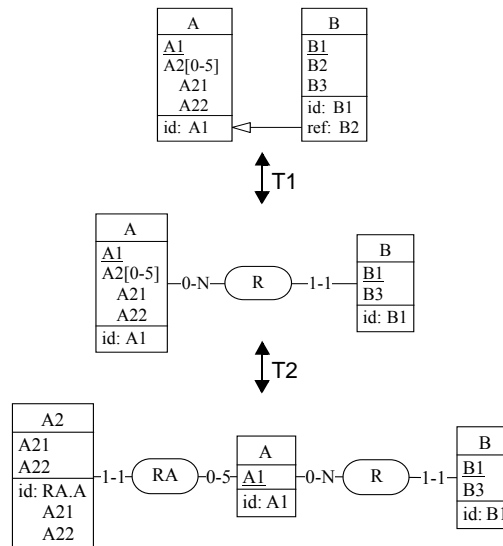


Figure 4-3: Sequence of two common (semantics-preserving) schema transformations: a foreign key transformation followed by an attribute transformation into an entity type.

4.3.5 Schema Integration

So far, we considered the mapping of one target to one source schema. Constructing one global from many source schemas, called schema integration, is formalized as schema transformation sequences between each source schema and the global schema (Figure 4-4).

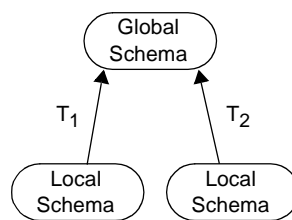


Figure 4-4: Schema integration as a set of schema transformation sequences.

4.4 Some Popular Transformations

We propose in Figure 4-5 and Figure 4-6 two sets of the most commonly used transformational operators. The first one is sufficient to carry out the transformation of most conceptual schemas into relational logical schemas. The second comprises additional techniques particularly suited to derive optimized schemas. Experience suggests that a collection of about thirty of such techniques can cope with most database engineering processes, at all abstraction levels and according to all current modeling paradigms¹.

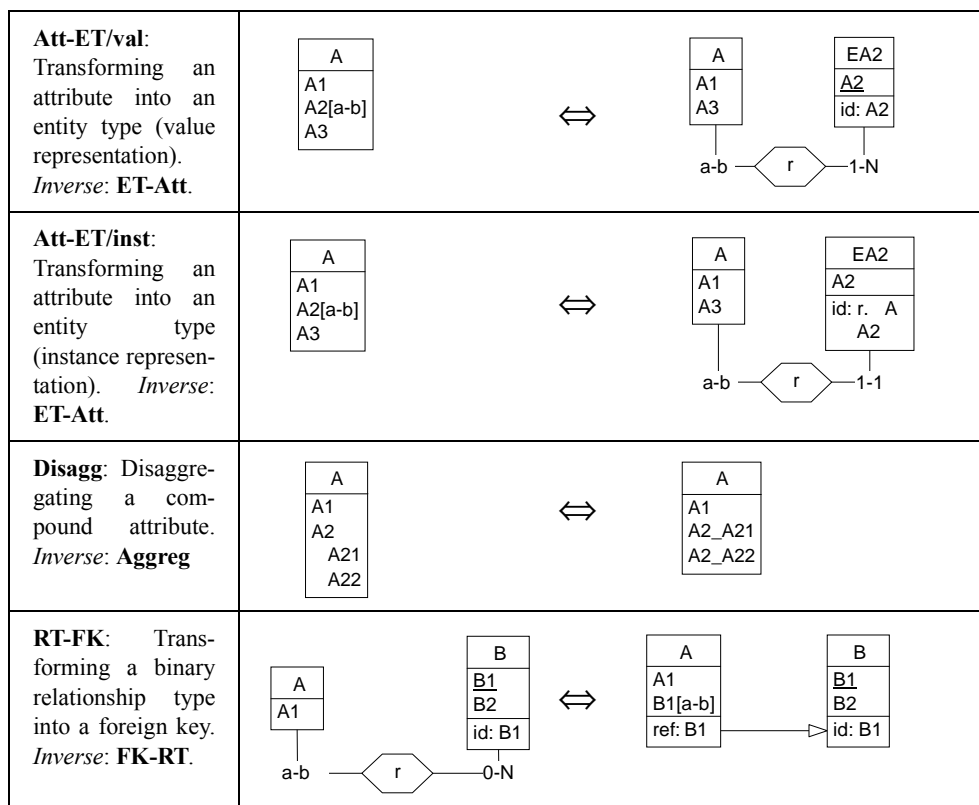


Figure 4-5: Four major generic transformations with their inverse. Cardinalities a, b, c and d must be replaced with actual values.

1. Provided they are based on the concept of record, entity or object.

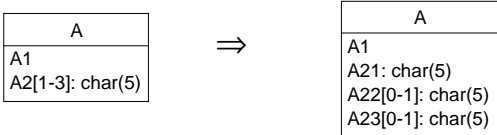
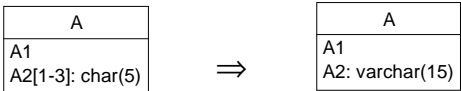
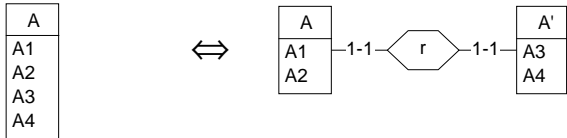
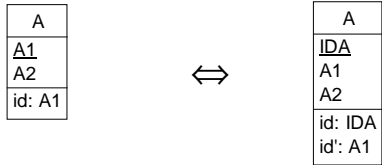
MultAtt-Serial: Replacing a multivalued attribute with a series of single-valued attributes that represents its instances. <i>Inverse:</i> Serial-MultAtt	
MultAtt-Single: Replacing a multivalued attribute with a single-valued attribute that represents the concatenation of its instances. <i>Inverse:</i> Single-MultAtt .	
Split: Splitting an entity type into two entity types. <i>Inverse:</i> Merge .	
AddTechID: A semantics-less attribute is added and made the primary ID of the ET. <i>Inverse:</i> RemTechID .	

Figure 4-6: Four additional generic transformations with their inverse.

4.5 Transformation History

The history of a schema transformation sequence is the recorded trace of all the transformations that are applied when transforming a schema *S* into a schema *S'*. Technically speaking, a history can be materialized by a sort of log file, and therefore is a pure sequence of transformation operations.

4.5.1 Structure of a History

A history can be available in different formats (Figure 4-7). We will describe three dimensions according to which histories can be classified.

Raw tree. An history is a sequence of transformation operations. If we consider multi-hy-

potheses approaches and decision processes, this sequence can be interpreted as a more complex graph. In general, a history has a directed acyclic graph structure, as illustrated in Figure 4-7a.

Linear history. Now, let us consider the successful branches only. We remove all the branches corresponding to hypotheses which have not been retained, and whose end products have been discarded. Keeping the live branches only produces a linear history (Figure 4-7b). This derived history is important since it describes the way the final products could have been obtained without any hesitation: replaying this history on the source products will yield the same output products as the actual process did.

Minimal linear history. Two branches of a linear history can represent the same type of transformations defined on the same object. For instance, a linear history can hold two naming transformations of a same attribute. Moreover, several branches of a linear history can represent transformations on the same object (type) that has been discarded in a next branch. Reducing these branches produce a minimal linear history (Figure 4-7c). This concept is interesting because it is the minimal form of a history.

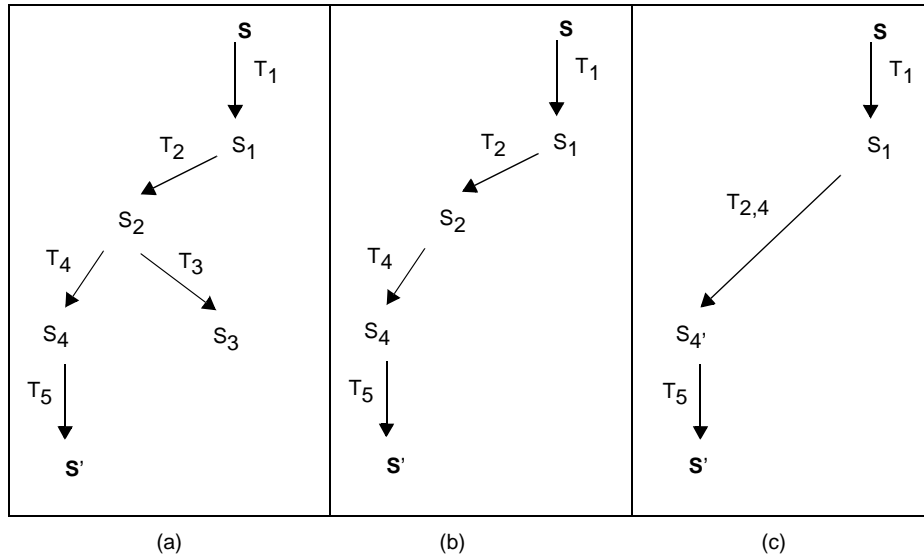


Figure 4-7: History Structures: (a) raw tree history; (b) linear history with the successful branches only; (c) minimal linear history.

4.5.2 History Subset

A history H_p is a *subset* of history H_n ($H_p \subseteq H_n$) if all the process instances of H_p appears in H_n , in the same order.

A history H can be sliced into sequences of transformations t_1, t_2, t_3 , etc. We will note this decomposition $H = \langle t_1 t_2 t_3 \dots \rangle$, where t_1, t_2, t_3 are transformations.

Let us consider history slice $h \subseteq H$, which starts at a time point where schema S_1 is known to be available. h can be seen as the history of a schema transformations, which produces product instance S_2 . We can write: $S_2 = h(S_1)$.

4.5.3 Independent Histories

Let us consider history $H_0 = \langle T_1 .. T_2 .. \rangle$, in which we identify transformations T_1 and T_2 . The question addressed is: does the execution of T_2 depend on the execution of T_1 , or are they independent, in which case they can be (or could have been) executed in any order, or even in parallel? First, we define the partial order relation $\text{before}(T_i, T_j)$, that states that slice T_i must be performed before T_j .

This relation is defined as follows:

$$\text{before}(T_i, T_j) \Leftrightarrow (C_+(T_i) \cap C(T_j) \neq \emptyset) \vee (C_0(T_i) \cap C_-(T_j) \neq \emptyset)$$

Intuitively, T_j must follow T_i if T_j uses constructs created by T_i , or T_j deletes catalytic elements of T_i . Then we define tr-before , the transitive closure of before

$$\text{tr-before}(T_i, T_j) \Leftrightarrow \text{before}(T_i, T_j) \vee (\exists t \subseteq T : \text{tr-before}(t_i, t) \wedge \text{before}(t, t_j))$$

Finally, T_1 and T_2 are *independent* iff

$$\neg \text{tr-before}(T_1, T_2) \wedge \neg \text{tr-before}(T_2, T_1)$$

4.5.4 Equivalent Histories

Two histories (or history slices) H_0 and H_1 are equivalent w.r.t. schema S iff

$$H_i(S) = H_j(S).$$

Let us consider history H_0 , which is expressed as a sequence of four subsequences:

$$H_0 = \langle h_1 h_2 h_3 h_4 \rangle,$$

where h_1 and h_4 are (possibly empty) sequences of transformations and h_2 and h_3 are two (non empty) history slices.

If we can prove that h_2 and h_3 are independent slices, then they can be swapped in H_0 , leading to history $H_1 = \langle h_1 h_3 h_2 h_4 \rangle$. Therefore, H_i is equivalent to H_j iff H_j can be built from H_i through a sequence of swap operations applied to independent slices.

Let us consider history H , which transforms the schema of Figure 4-8 into a relational schema: multivalued attribute **DETAIL** is transformed into entity type **DETAIL** and one-to-many **rel-type** **from**, then the latter and **rel-type** are expressed as foreign keys.

$H :$ $h_1: (\text{DETAIL}, \text{from}) \leftarrow \text{Att-to-ET/Value}(\text{ORDER}, \text{DETAIL})$

$h_2: \{\text{ORD-ID}\} \leftarrow \text{RT-to-FK}(\text{from}, \text{DETAIL})$

$h_3: \{\text{ORD-ID}\} \leftarrow \text{RT-to-FK}(\text{of}, \text{PRODUCT})$

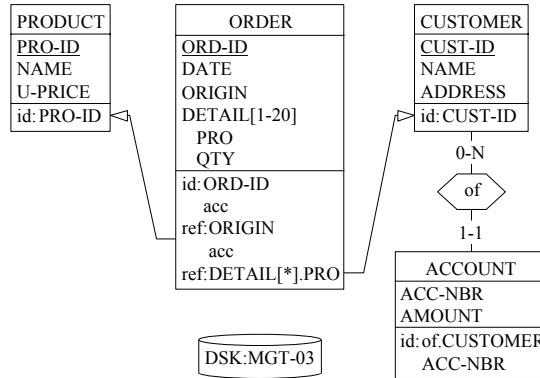
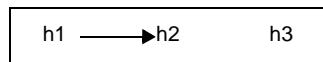


Figure 4-8: A schema example.

The graph of tr-before is as follows :



Therefore, $\langle h2, h3 \rangle$ and $\langle h1, h3 \rangle$ are independent and swappable. According to the definition, $\langle h3, h1, h2 \rangle$ and $\langle h1, h3, h2 \rangle$ are equivalent to H , while $\langle h3, h2, h1 \rangle$ is not equivalent.

4.5.5 Minimal History

The history of a design process records the results of the decisions, be they right or wrong, of trials, errors, backtracking, undos and redos which shape all the exploratory human activities. Histories generally have a complex structure including several branches which materialize the exploration of concurrent hypotheses, of which one only led to the discovery of a target concept, the other ones being abandoned. Cycles of doing, then undoing, and finally redoing, are not uncommon either. Such structures must be simplified: multiple branches must be reduced to the only one that has proved useful, useless loops must be discarded. Hence the concept of minimal history, which can be defined as follows:

history H is *minimal* w.r.t. schema S iff for any $T_p \subset T_n$, $T_p(S) \neq T_n(S)$

In other words, there is no proper subsets of H which still are equivalent to H . Given history H , H_m is a minimal version of H if H_m is minimal, and H_m is equivalent to H .

4.6 Model Transformation

The *model translation* is a particular case of schema transformation. It consists in translating a schema expressed in a data model M_s into a schema expressed in another data model M_t . We use the model translation concept to illustrate those of schema transformation and transformation history.

The model translation is defined as a *model-driven transformation* within the generic model. A model-driven transformation applies on a schema. It can be defined by $m(M_s, M_t)$ where M_s and M_t are two different submodels, i.e., subsets of the generic model. It consists in applying the relevant transformations on the relevant constructs of the schema expressed in M_s in such a way that the final result complies with M_t .

A model-driven transformation is expressed as a *transformation plan* made up of a sequence of <condition, action> statements and control structures, where condition is a structural predicate and action is a transformation. The meaning is obvious: apply action on each object that satisfies predicate condition. The control structures include scope restrictions and loops.

As an illustration of model translation, we consider the transformation plan between the wrapper logical model and the object-oriented model defined in Chapter 3. That is, a schema expressed in the wrapper logical model (M_s) is translated into a schema expressed in the object-oriented model (M_t).

- 1- While compound attributes exist do:
 - For each attribute A that is single-valued and that depends directly on an entity type E do:
 - apply** Att-ET/inst to A;
 - add** a process unit P to E; // name of the process unit: *getAttributeName*
 - For each attribute A that is multivalued and that depends directly on an entity type E do:
 - apply** Att-ET/inst to A;
 - add** two process units P_1 and P_2 to E;
 - // names of the process units: *getFirstAttributeName* and *getNextAttributeName*
- 2- For each referential attribute A of an entity type ET_s that references another entity type ET_t , do:
 - apply** FK-RT to A;
 - add** a process unit P to ET_s ; // name of the process unit: *getETsname*
 - add** two process units P_1 and P_2 to ET_s ;
 - // names of the process units: *getFirst ETtName* and *getNext ETtName*
- 3- For each entity type E do:
 - For each single-valued attribute A of E do:
 - add** a process unit P to E; // name of the process unit: *getAttributeName*
 - For each multivalued attribute A of E do:
 - add** two process units P_1 and P_2 to E;
 - // names of the process units: *getFirstAttributeName* and *getNextAttributeName*

This transformation plan can be applied to any schemas expressed in the wrapper logical

model. Its execution produces two result types: (1) a target schema expressed in the object-oriented model and equivalent to the source schema; and (2) a schema transformation history that records all the transformations applied by the transformation plan.

Example

Let us consider the wrapper logical schema of the Figure 4-9.

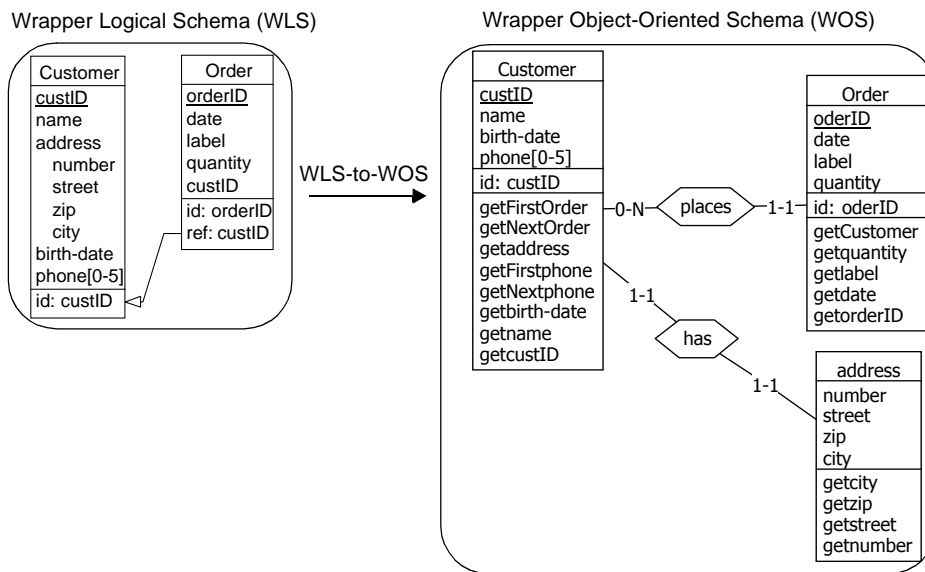


Figure 4-9: Model translation of a wrapper logical schema into an object-oriented schema.

The transformation plan application on this schema is translated into its history:

- We express the compound attribute address as an entity type address and we add the corresponding process unit to the entity type Customer.
 T1: (address, has) \leftarrow Att-ET/inst(Customer, address)
 T2: (getFirstAddress) \leftarrow AddProcessUnit(Customer)
- The schema includes one referential attribute (custID of the entity type Order) which expresses a relationship types. We augment the history with the following transformations:
 T3: (places) \leftarrow FK-to-RT(Order, {custID}, Customer)
 T4: (getFirstOrder) \leftarrow AddProcessUnit(Customer)
 T5: (getNextOrder) \leftarrow AddProcessUnit(Customer)
 T6: (getCustomer) \leftarrow AddProcessUnit(Order)

- Finally, we allocate the process units to all the entity types of the schema:

T7: (getCustID) \leftarrow AddProcessUnit(Customer)
 T8: (getName) \leftarrow AddProcessUnit(Customer)
 T9: (getBirth-date) \leftarrow AddProcessUnit(Customer)
 T10: (getFirstPhone) \leftarrow AddProcessUnit(Customer)
 T11: (getNextPhone) \leftarrow AddProcessUnit(Customer)
 T12: (getOrderID) \leftarrow AddProcessUnit(Order)
 T13: (getDate) \leftarrow AddProcessUnit(Order)
 T14: (getLabel) \leftarrow AddProcessUnit(Order)
 T15: (getQuantity) \leftarrow AddProcessUnit(Order)
 T16: (getNumber) \leftarrow AddProcessUnit(Address)
 T17: (getStreet) \leftarrow AddProcessUnit(Address)
 T18: (getZip) \leftarrow AddProcessUnit(Address)
 T19: (getCity) \leftarrow AddProcessUnit(Address)

The history of the transformation plan of the wrapper logical schema (WLS) of the Figure 4-9 records the transformation sequence WLS-to-WOS:

WLS-to-WOS = $\langle T19 \circ T18 \circ T17 \dots \circ T3 \circ T2 \circ T1 \rangle$

The wrapper object-oriented schema of the Figure 4-9 (WOS) is then obtained by the application of the transformation sequence WLS-to-WOS on WLS:

WOS = WLS-to-WOS(WLS)
 or WOS = T19(T18(T17(...(T3(T2(T1(WLS)))))))

Part II

Wrapper Technology

Wrapper Architecture

In which the technology of wrappers for legacy databases is presented. Their characteristics are outlined and a definition of legacy data wrappers is proposed. The different levels of services a legacy data wrapper should manage are presented and discussed. The architecture of an operational data wrapper - the InterDB wrapper - is finally presented.

5.1 Introduction

Legacy data systems contain very valuable information that is embedded in legacy databases/flat files and application code [Umar, 1997]. In many cases, legacy data systems are the only source of years of business rules, historical data, and other valuable information. Access to this information is of vital importance to new and emerging tools and applications.

A *wrapper* attempts to extend the usefulness of the legacy data systems by facilitating their integration into modern (distributed) systems. As systems age, the knowledge base that created them fades. Eventually, the needs that the system addressed change, and, therefore, the system must be changed. Here, the analysts have three options available to them: modify the system (and potentially cause its failure); create a new system with the new functionality; or keep the old system and create a new layer, a wrapper between the legacy system and the new program interface [Rugaber, 1998].

The approaches of wrapping legacy data systems can be divided into three main categories: (1) encapsulating screens (2) encapsulating the procedural components and (3) encapsulating the data. The objective of this paper is to contribute to the building of the third kind of wrappers. However, it is worth discussing the goal and problems of these approaches.

Wrapping screens

Screen scrapers allow client application to simulate the terminal keyboard/display features and thus act as programmable terminal emulators [Umar, 1997]. Commercially available (e.g., Computer Associates), screen scrapers are inexpensive and are frequently used to access legacy systems that are not well structured (monolithic). In many old systems, terminal emulation and screen scraping are the only ways to access legacy data.

Wrapping procedural components

According to the second interpretation (e.g., [Gall, 1995], [Sneed, 1997], [Wiggerts, 1997]), a wrapper consists in encapsulating potential object classes and their basic methods. A legacy application is analyzed in order to build a description of its data objects and as many as possible parts of its procedural components. For example, a COBOL business application based on files *Customer*, *Item* and *Order* will be given a description comprising *Customer*, *Items* and *Order* classes, with their associated methods such as *RegisterCustomer*, *DropCustomer*, *ChangeAddress*, *SendInvoice*, etc.

Unfortunately, building such wrappers is a very (too?) complex task. Indeed, the process of code analysis must take into account complex patterns [Sneed, 1995] such as code replication (near-identical code sections duplicated throughout the programs); common usage (a single code used by several sections) and runtime determined control structures (e.g., dynamically changing the target of a *goto* statement or dynamic SQL).

Several code analysis techniques have been proposed to examine the static and dynamic relationships between statements and data structures (e.g., [Cimitile, 1998], [Henrard, 1999]). Dataflow graphs, dependency graphs and program slicing are among the most popular.

Wrapping data

The third approach (e.g., [Papakonstantinou, 1995], [Vidal, 1998]) proposes to leave the code and the screens aside and to wrap the data only. The legacy data system is accessed via standard object definitions without disorganizing the legacy data. The problem is of course different, and fortunately a bit easier than application wrapping:

- the semantic distance between the so-called conceptual specifications and the physical implementation is most often narrower for data than for procedural parts (a COBOL file structure is easier to understand than a COBOL procedure);
- the permanent data structures are generally the most stable part of applications;
- even in very old applications, the semantic structures that underlie the file structures are mainly procedure-independent, though their physical structure is highly procedure-dependent.

5.2 Legacy Data Wrapper Definition

5.2.1 Overview

The wrapper architecture and interfaces are crucial, because wrappers are the focal point for managing the diversity of data sources. Below a wrapper, each data source, or data management system, has its own data model, schema, programming interface, and query capability. The data model may be relational, object-oriented, or specialized for a particular domain. The schema may be fixed, or vary over time. Some data systems support a query language, while others are accessed using a class library or other programmatic interface. Most critically, legacy data systems vary widely in their support for queries. At one end of the spectrum are legacy data systems that only support simple scans over their contents (e.g., files of records). Somewhat more sophisticated data systems may allow a record ordering to be specified, or be able to apply certain predicates to limit the amount of data retrieved. At the other end of the spectrum are databases like relational databases that support complex operations like joins or aggregations.

5.2.2 Definition

A *legacy data wrapper* is a converter of a legacy DMS interface (query, model, services and protocol). More specifically, a *legacy data wrapper* is a software component that is built on a legacy DMS and offers a new interface to the legacy data managed by it without modifications of the legacy database (e.g., preserving the legacy database schema and behavior).

In that way, wrappers can be used to present a simplified DMS interface, to encapsulate diverse heterogeneous data systems so that they all present a unique query interface based on a common data model, to add functionality to the data legacy system (e.g. security management), or simply to open a legacy data system to other systems.

5.2.3 Functionality

The main functionality of a data wrapper can be subdivided into a number of categories:

- *Generic functionality.* This refers to operations that convert data and queries from one model to another. Typically, wrappers convert queries into one or more commands/queries understandable by the underlying legacy system and transform the results into a format understandable by the new application.
- *Source-specific functionality.* This refers to the operations that take place because of the contents of the legacy data system, i.e., its schema.
- *Control structure.* A wrapper can change the way in which the requests and responses are passed. For example, a synchronous data source may be wrapped by an asynchronous wrapper that buffers responses until they are requested by the caller.

- *Error reporting.* Wrappers can report errors back to the caller. These may be error messages generated by the data source, perhaps converted to a desired format, or they can be the results of an error recovery operation within the wrapper.
- *Security.* Data security is another important function of a wrapper that protects data against unauthorized access through its interface. Data security includes two aspects: data protection and authorization control.
- *Concurrency and recovery control.* This function is to permit concurrent updates of the underlying legacy databases. This includes the transaction and failure management.
- *Semantic integrity control.* Wrappers emulate integrity constraints defined at their interface but not declared in the underlying database definition.

5.2.4 Legacy Issues

Wrapping legacy data systems poses complex problems. In this section, we discuss some important specificities of a legacy data system: (1) its autonomy; (2) its data model, (3) its physical schema, (4) its access language. We discuss also the requirements of a wrapper which is intended to encapsulate such a system.

Legacy data system autonomy

Organizational entities that manage different database systems are often autonomous [Elmagarmid, 1999]. In other words, databases are often under local and independent control. Those who control a legacy data system are often willing to keep the legacy applications access without modify them. Thus, it is important to understand the aspects of autonomy and their influence in the wrapper development.

DMS autonomy. Keeping DMS autonomy assumes that the legacy DMS retains complete control over data and processing. The wrapper can only interact with its underlying database through the legacy DMS external interface. As a result of DMS autonomy, some internal information, such as local cost parameters, needed for query optimization are not available for the wrapper.

Legacy application autonomy. Legacy application autonomy means that the wrapper can not influence the way whose individual legacy applications access to data. It means that the wrapper can't interfere with their execution. In other words, the wrapper must be aware that it isn't alone accessing the legacy data.

Legacy data models

Legacy data systems are based on various legacy data models, ranging from standard file to relational model. To deal with such a kind of heterogeneity, a wrapper should hide the model that a legacy system implements by providing a more abstract and common model. Such a model must be highly generic and more flexible than the legacy data models [Garcia, 1997].

Legacy physical schemas

Legacy data models can not express all the semantics of the real world. Limitations of the modeling concepts lead to the incompleteness of the physical schema [Parent, 1998]. A wrapper can't therefore assume the quality and the completeness of the physical schemas. The wrapper must offer a semantically rich description of a legacy data system. Extracting a semantically rich description from a data source is the main goal of the data-centered reverse engineering process (DBRE). Hence, the close link between the *reverse engineering* and the wrapping [Thiran, 2001].

Legacy access languages and services

Different languages are used to manipulate data represented in different data models. The query capabilities of these languages are multiple and various (e.g., COBOL program vs. SQL). A wrapper manages the translation of commands (e.g., queries) from one language (e.g., query language) to another. This is not always possible to translate a wrapper query into a single legacy query. This is due to some legacy query language limited functionality (e.g., some operations required by wrapper query are not supported by the local legacy data system).

The assumption that DMS are equal in terms of their processing capability is not applicable since legacy DMS may vary drastically in terms of their availability and processing costs. For instance, a legacy DMS may lack important features like transaction management. Semantic heterogeneity may have an adverse impact on query processing. The query processor of a wrapper needs to be aware of legacy DMS query capabilities in order to use them in an efficient way or to avoid unnecessary computation. Therefore, it is important that the DMS query capabilities are taken into consideration by the wrapper developer.

Example

To illustrate, consider an application that issues a query to a wrapper. The underlying legacy information data it accesses are recorded in COBOL files.

1. Assume the following query processed in the wrapper: Select c.name from customer c where c.custCode = 'HTB710'. The wrapper processes the above query by transforming it into a COBOL program in taking the physical characteristics of the COBOL files into account. For instance, if it is defined as a record key in an indexed file, the wrapper should process an indexed access instead of a sequential one.
2. Assume the following query: Select c.name from customer c, order o where c places o; The underlying legacy system doesn't understand the notion of relationship. Hence, the wrapper should simulate the query by transforming it into an understanding way for COBOL.
3. Moreover, if the wrapper offers the transaction functionality, it should manage the ACID properties since COBOL doesn't support all these properties.

In the first example, the wrapper must take into account the DMS and physical characteristics of the underlying DMS. In the second and third examples, the wrapper must simulate all the concepts and functions that aren't supported by it. As we can see, the wrapper should exploit both the features of the DMS (query language and data model) and the features of a particular data structure (physical schema).

5.2.5 Motivations and Objectives

Motivations

In general, wrapping is a very attractive strategy due to several reasons. First, it extends the useful life of the legacy data systems by making them useful to new applications without modifications to the legacy applications. Second, it addresses the challenge of database heterogeneity by providing a standardized and common interface.

Objectives

Legacy data wrapping is generally intended to satisfy the following objectives:

- *Legacy system migration.* Migrating a system consists in replacing one or several of the implementation technologies. IMS/DB2, COBOL/C, monolithic/client-server, centralized/distributed are some widespread examples of system migration. In some situations, the only component to salvage when abandoning a legacy system is its database. The data have to be converted into another format, which can be taken in charge by a wrapper.
- *Legacy data extraction/conversion.* Most datawarehouses are filled with aggregated data that are extracted from corporate databases. This transfer requires a deep understanding of the physical data structures and of their semantics, to write a wrapper layer that interprets them correctly.
- *Legacy system extension.* This term designates changing and augmenting the functional goals of a legacy system, such as adding new functions, or its external behaviour, such as improving its robustness or its security.
- *Legacy system opening to modern system.* Legacy systems provide service that remain useful beyond the means of the technology in which they were originally implemented (they are often well-suited to run certain applications, such as high-volume batch processing operations). The wrapper attempts to extend the usefulness of those systems by facilitating their communication with other modern (or legacy) systems.
- *Legacy system integration.* The wrapper is one of the architecture component of a database federation (see Chapter 1).

Technical issues

Wrapping legacy data introduces several technical issues that fall into the following broad

categories:

- *Wrapper complexity.* Wrapping legacy data systems can be an expensive undertaking especially when it involves that the wrapper integrates much complex functionality.
- *Wrapper maintenance.* A wrapper has to deal with upgrade versions of the legacy DMS and the evolution of the underlying database.
- *Legacy access performance.* The addition of yet another layer can introduce performance delays.
- *Legacy system keeping.* Wrappers don't solve the root problem. Underneath a wrapper, the legacy data system remains, aging and inflexible. A wrapper only offers a way to delay the expense of a complete rewrite by providing an acceptable interface for new applications.

5.3 Architecture

5.3.1 General Framework

The basic idea is that the legacy databases are not touched; instead they are surrounded by wrappers. Figure 5-1 shows the general framework that overviews the different aspects of wrapping legacy databases. The framework consists of the following components:

- *Legacy applications* that already exist and access to a legacy database;
- *Access paradigms* that are used to remotely access legacy resources;
- *Wrapper technology* that attempts to hide the characteristics of a legacy database from the new applications;
- *New applications* that access to a legacy database through a wrapper.

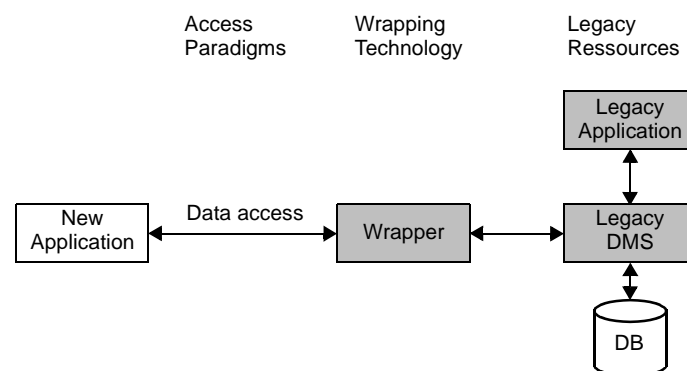


Figure 5-1: Legacy data wrapper framework.

Legacy resources

Legacy resources contain information that is embedded in legacy databases/flat files and legacy applications. They are often built around a legacy database service (e.g., IMS). Sometimes they don't use a DMS at all. Instead, the legacy resources are based on flat-file structures of ISAM, VSAM or other file systems.

Accessing to legacy information through a wrapper implies a certain level of legacy resources friendliness because the legacy data must be exposed to external (and new) applications.

Access paradigms

Access infrastructure consists of technologies such as computers, networks and transaction managers. An important part of the platform is *middleware*, an increasingly crucial and, at the same time, bewildering component of the access infrastructure. Middleware is needed to interconnect and support applications of the modern access infrastructure. Middleware services typically include directories, facilities to call remotely located functions and software to access and manipulate remotely located databases. Middleware services are typically provided by specialized software package. However, middleware services may reside in a combination of database management systems, computer operating systems, and transaction management system.

The middleware manages the communications between the wrapper and client applications. CORBA [Mowbray, 1995] and RMI [Reese, 1997] are two examples of middleware technologies. CORBA and RMI are two distributed object standards supported by the OMG (Object Management Group). CORBA is an architecture standard for building heterogeneous distributed systems. RMI supports distributed objects written entirely and only in the JAVA programming language. For the future, it's expected to enable RMI to use the IIOP protocol to communicate with CORBA-compliant remote objects.

By using CORBA, it is possible to encapsulate a wrapper as a set of distributed objects and their associated operations [Dogac, 1995]. These properties provide the means to handle the heterogeneity at platform and location levels, the semantic heterogeneity being solved by the wrapper. That is, CORBA allows client applications to communicate with a wrapper without having knowledge of its location.

Moreover, CORBA defines an interface definition language that makes it possible to define user-defined objects that represent conceptual views of the legacy data. By defining these user-defined objects, the semantics understanding can be enforced. Indeed, these user-defined objects play a central role in capturing the semantics of actual needs, in a way that is very closed to the business reality [Maniola, 1998].

New applications

New applications are the software components that access to the legacy database through the wrapper interface. A new application can be, among others, a mediator or an extract-transform-load processor. The type of the new application strongly influences the wrapper build-

ing. For instance, update wrappers are often useless in migration process because the wrapper is only used as a data extractor.

5.3.2 Wrapper Interface

A wrapper is developed on top of a legacy database to give it a transparent access interface. The interface is made up of: (1) a *wrapper schema* of the wrapper database, expressed in a canonical data model and (2) a *common query language* which uses the semantics defined in the wrapper schema. That is, the wrapper interface is often required to handle [Lim, 1999]:

- *Exporting and homogenizing the schemas of the legacy sources.* Wrapper schemas and data of existing databases to the new applications. The set of schemas and data exported from the existing database is called a *wrapper schema*. Hence, the definition of the conceptual schema is sometimes called semantic enrichment. However, it is equally possible that some local semantics are not included in the wrapper schema.
- *Homogenizing the query interface to existing databases.* Apart from exporting local databases, wrappers have to process queries on the wrapper schemas. Queries on the wrapper schemas are also known as *wrapper queries*.
- *Controlling the subset of local database accessible by the new applications.* Another important reason for having wrapper schema is to control the subsets of local databases (both in terms of schema structures and instances) accessible by the new applications.

Schema interface

The wrapper schema is the result of both a *model translation* and a *semantic enrichment* process (see Chapter 6). The physical schemas expressed in local data models are translated into wrapper schemas expressed in a canonical data model. Moreover, since we assume that the underlying databases are legacy systems, the wrapper schema also contains and incorporates extra semantics that are not found in the physical schema. That leads to get a semantically rich description of the databases.

Example

Let us consider a wrapper defined for relational databases and that offers an OO interface (Figure 5-2). Through the model translation and reverse engineering processes, the physical schema made up of two files is translated into a wrapper schema using the entity-object relationship model. The wrapper schema includes undeclared constructs like the multivalued and compound attributes (*Detail*) and implicit constraints like the relationship between the two object types.

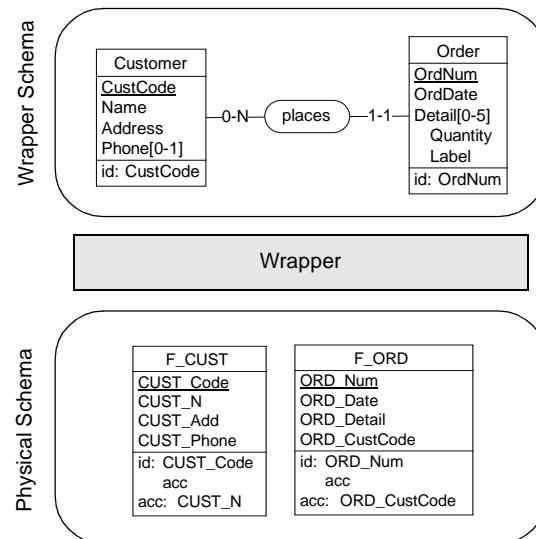


Figure 5-2: Physical and wrapper schema example.

Query interface

The query language of a wrapper allows writing queries expressed on wrapper schema constructs. The wrapper queries therefore address the legacy data independently of the specific aspects of a family of models as well as of the technical constructs of the actual database. Functionally, the wrapper translates the queries expressed on wrapper schema objects into commands expressed on physical schema constructs. It also assembles the extracted physical data (records and rows) into wrapper objects (see Section 5.4.4).

Example

Let us consider the wrapper example of Figure 5-2. The wrapper query showed in the upper part of Figure 5-3 uses the semantics of the wrapper schema. This query is translated in a SQL query expressed on the physical schema.

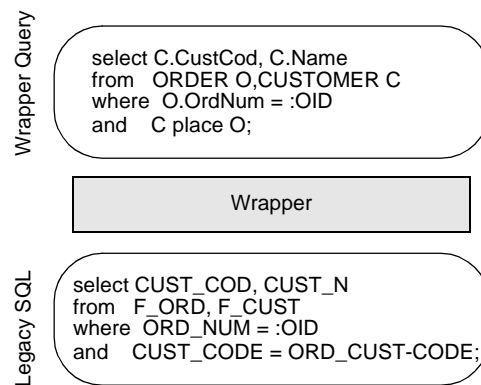


Figure 5-3: Query translation example.

Query and data mappings

One major wrapper service is the *data and queries translation* from one model (the legacy DMS model) to another, the canonical data model. That is, it translates queries expressed on the canonical data model into commands expressed on physical constructs; and conversely, it assembles extracted physical data into output objects. For instance, a wrapper associated with a set of COBOL files translates the input queries into COBOL program codes and assembles the COBOL records into output objects.

If the wrapper allows updates, it must also ensure the consistency of the legacy data. This can lead the wrapper to emulate *advanced services* such as integrity control and transaction and failure management if the underlying DMS doesn't support them.

Figure 5-4 shows the successive steps of query translation and result formation.

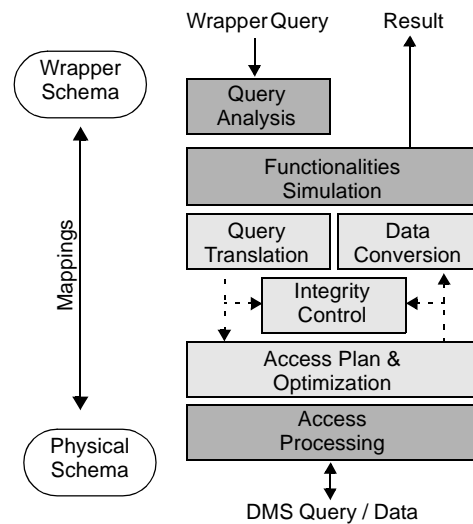


Figure 5-4: Service-oriented wrapper architecture.

5.4 Wrapper Services

Figure 5-5 shows the major services of a legacy data wrapper, namely, the query analysis, the error reporting, the query processing, the semantics integrity control and the functionality emulation. To provide these services, a wrapper can advantage of whatever specialized services the legacy DMS already provide without modifying the legacy database structure and behavior. We explain and discuss all these services in the next sections.

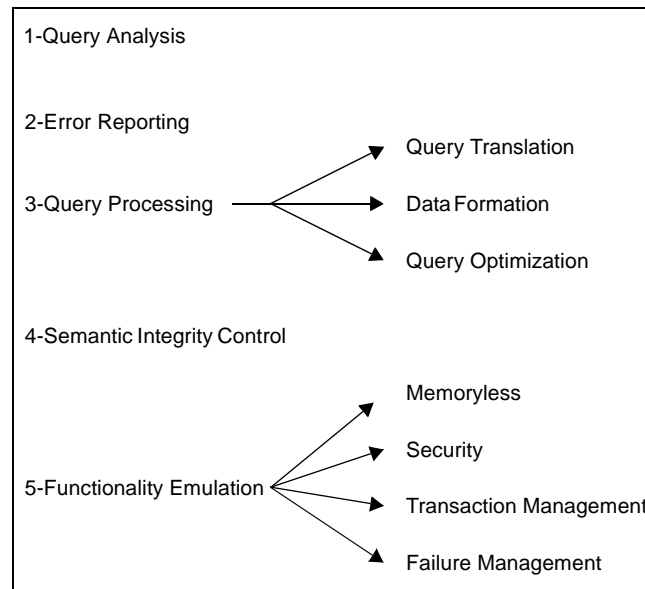


Figure 5-5: Main services of a legacy data wrapper: (1) query analysis; (2) error reporting; (3) query processing; (4) semantic integrity control; and (5) functionality emulation.

5.4.1 Query Analysis

The first task of a wrapper is the analysis of the input queries. Query analysis enables rejection of queries for which further processing is either impossible or unnecessary [Özsu, 1991]. The main reasons for rejection are that the query is syntactically or semantically incorrect. When one of these cases is detected, the query is simply returned to the user with an explanation (see Section 5.4.2). Otherwise, query processing is continued. A query is incorrect if any of its attribute or entity type names are not defined in the wrapper schema, or if operations are being applied to attributes of the wrong type.

Example

The following query on a wrapper that offers a SQL-like query language and the wrapper schema of the Figure 5-2:

```
SELECT Num FROM Customer WHERE Name = 2;
```

is incorrect for two reasons. First, attribute Num is not declared in the schema. Second, the operation =2 is incompatible with the type string of Name.

Query analysis can be viewed as three successive steps. First, the query is syntactically analyzed. Second, the query is analyzed semantically so that incorrect queries are detected and re-

jected as early as possible. Techniques to detect incorrect queries use some sort of graph that captures the semantics of the query. Third, the correct query is restructured as an internal representation (see Section 5.4.4).

5.4.2 Error Reporting

A wrapper returns a value that indicates the success or the failure of an input query. An error can occur at two levels:

- *At the legacy DMS level:* most legacy DMS returns some indicators of a query execution. This indicator is often made up of two parts: the error code and an associated message. A legacy database errors can result from anyone of many problems such as conversion errors, arithmetic errors, constraint violation, etc.
- *At the wrapper level:* the wrapper can also catch other errors than those returned by the underlying DMS. For example, a wrapper can detect an error when it performs a query analysis (see Section 5.4.1) or a semantic control of implicit constraints (see Section 5.4.5).

Besides the error codes it detects, a wrappers must provide standardized error codes of DMS-specific errors to give new applications a standard way of dealing with error conditions. Although DMS return similar kinds of errors, each does in a different manner, using different error numbers, message types, programming styles. A wrapper must therefore simplify dealing with error information by providing:

- *A unified return code* mechanism that reports success or failure for each data access whatever the source (DMS or wrapper);
- *A standardized error code.* A standard error code can be, for instance, the five-character sequence defined by the ISO SQL-92 standard.

5.4.3 Functionality Emulation

Due to the limited functionality of some DMS, the wrapper must often simulate operations and behaviours required by the new applications. Their nature depends on the differences between the functions provided by the DMS and those required by the new applications. That's the way a wrapper may provide a wide range of functions such as the following: memoryless, security, transaction management, failure management.

Memoryless

A *context handle* holds all the information required by the wrapper for performing a request. An instance of the context handle is associated with one application client connection. The instance retains state on behalf of the client across multiple request invocations. It is used to identify a client application and to keep tracks of what has happened between this client and the wrapper.

A *memoryless wrapper* is a wrapper that keeps no context handle of requests that it performs. Consequently, memoryless wrapper can be restarted after a failure without any need to restore any state.

Memoryless wrappers are not required to maintain contextual information about the client applications. Each request from a client application contains the context handle needed to satisfy the request¹. To submit a query, an client application sends its request and its context handle to the wrapper. The wrapper uses the context handle to identify the client and prepares the request processing. It updates the context handle according to the request results and then returns it to the client application.

Security

Definition and concept evolution. Security is a method to maintain accountability and control access to the system resources. In legacy centralized data environments, both programmers and users of the legacy system were trusted implicitly, because physical access to the computing center was required to access them. As systems became distributed, physical access was no longer required to the system [Souder, 2000]. In place of the original physical access controls, software security was introduced to the systems.

Since these early access models were an extension of the original physical security models, users were granted trusted access to a host. This created problems when distributed systems were introduced that granted trust to hosts rather than the individual users.

Similarly, the view of a user evolved from an entity given a high level of trust to an entity with assumes a set of roles. In this model, the roles are granted access to the system. Thus, the users who perform those roles are granted access for only the activities related to those roles. *Role Based Access Control* (RBAC) entered the field of computer security [Sandu, 1996].

In a distributed environment system, the data encapsulated within the systems became distributed. Data distribution generally transfers it across a secondary medium (e.g., the Ethernet cable) between nodes in the system. In this secondary medium, data are now publicly available to anyone who has physical access to the medium. Hence, data must be protected in transit. *Encryption* is designed to provide such a data protection [Rushby, 1983].

When a legacy data system is wrapped, it is commonly designed to provide a secure access. Before the legacy data system is wrapped, the legacy system (DMS and OS) defined which users were permitted to access its services through a locally-defined security system. When the legacy data system is wrapped, the distributed environment imposes its own authorization control and data protection. Hence, the legacy data system sits between the legacy security and the distributed security systems.

Security and wrapper. As a consequence, security managed by a wrapper includes two as-

1. Note that the context handle is hidden to the client applications. Client applications never look "inside" a handle and cannot directly manipulate the contents of the context handle.

pects: data protection, authorization control [Özsu, 1991].

- *Data protection* is required to prevent unauthorized users from understanding the physical content of data. This function is typically provided by file systems in the context of centralized and distributed operating systems. The main data protection approach is data encryption which is useful for information exchanged on a network.
- *Authorization control* must guarantee that only authorized users perform operations they are allowed to perform on the database. Many different users may have access to a large collection of data under the control of a single centralized or distributed system. The wrapper must thus be able to restrict the access of a subset of the legacy data to a subset of the users. A wrapper providing a security layer has been developed at the Drexel University [Souder, 2000].

Example

An authorization can be viewed as a triple <user, operation type, object definition> which specifies that the user has the right to perform an operation on an object. To control authorizations properly, the wrapper requires users (pairs of user name and password), objects and rights to be defined. The privileges of the users over objects are recorded in a directory as authorization rules, managed by the wrapper. There are several ways to store the authorizations [Özsu, 1991]. The most convenient approach is to consider all the privileges as an *authorization matrix*, in which a row defines a subject, a column an object, and a matrix entry (for a pair of <subject, object>), the authorized operations. The authorized operations are specified by their operation type (e.g., SELECT, UPDATE). Figure 5-6 gives an example of an authorization matrix where objects are either entity types (Customer) or attributes (Name).

Users	Customer	Name
pth	SELECT	SELECT
jlh	SELECT	UPDATE
jmh	NONE	NONE

Figure 5-6: Example of authorization matrix.

Example

The wrapper schema represents the part of a legacy database that is provided by the wrapper. A wrapper schema can therefore be used to hide sensitive data from unauthorized users. This works like the SQL views [Date, 1995].

Transaction management

Transaction management is probably the major open question in wrapper systems. The challenge is to permit concurrent updates to the underlying legacy systems without violating their

autonomy. Although this subject is somewhat beyond the scope of this report, we discuss it briefly for sake of completeness. Transaction management can be viewed in two dimensions: autonomy and heterogeneity.

Autonomy. It requires that the transaction management functions of a wrapper be performed independent of the DMS transaction management execution functions. In other words, the DMS are not modified to accommodate wrapper updates.

Heterogeneity. It has the additional implication that the wrapper transaction managers of each DMS family may employ different concurrence control and commit protocols. Heterogeneity adds further difficulty since it becomes difficult to make uniform assumptions about the functionality provided by legacy DMS. However, if a legacy DMS has techniques that enable concurrent and recoverable access to local data source, the wrapper can use them with a minimal effort. Some old DMS don't support any commit protocol. Most recent DMS commit protocols contain some operators, for instance *begin*, *commit* and *abort*, that allow the users to mark the code that is implied in a transaction. Other DMS, defined for more advanced commit protocols, include more behavior. For instance, in order to use the two phase commit protocol (2PC) [Gray, 1993], a *prepare_to_commit* operator must be included. Moreover the semantics of the same operator can change from one commit protocol to another. For instance, in the flat transaction model [Gray, 1993], the *commit* operator leads to make visible for every other transactions the effects of the current transaction. In the nested transactional model [Moss, 1985], however, the *commit* operation leads to make visible only for the ancestors and sisters of the current transaction.

Failure management

A reliable wrapper is one that can continue to process user requests even when the underlying database is unreliable. In other words, even when components of the legacy DMS fail, a reliable wrapper should be able to continue executing user requests without violating database consistency.

The two fundamental approaches to constructing a reliable wrapper are fault tolerance and fault prevention. *Fault tolerance* refers to a wrapper which recognizes that faults will occur; it tries to build mechanisms so that the faults can be detected and removed or compensated. *Fault prevention* techniques aim at ensuring that the wrapper system will not cause any faults. Fault prevention has two aspects. The first is *fault avoidance*, which refers to the techniques used to make sure that faults are not introduced into the legacy system by the wrapper. These techniques involve detailed design methodologies and quality control of the legacy system. The second aspect of fault prevention is *fault removal*, which refers to the techniques that are employed to detect any faults that might have remained in the legacy system despite the application of fault avoidance and removes these faults. Typical techniques that are used in this area are extensive testing and validation procedures.

Designing a reliable wrapper that can recover or prevent the failures requires identifying the types of failures with which the wrapper has to deal. A detailed review of the major reasons

of failures and a discussion of the wrapper reliable design are beyond the scope of this report. A review of the major failures appears in [Ozsü, 1991].

Example

With respect to the fault tolerance, a memoryless wrapper is a system that recognizes that a system failure can occur: such a wrapper can be restarted after a failure without any need to restore any state.

5.4.4 Query Processing

Query processing involves several main steps. A high-level query such as SQL or OQL is first scanned, parsed, and validated by a parser (Cfr. Section 5.4.1). Then, an internal representation for the query is built. Resulting representation is then sent to a query optimizer. The *query translator* is the core component of the query processor. It has to translate a query based on the canonical data model and query language into a query understandable by the underlying DMS.

Query processing problem

The main function of the wrapper query processor is to transform a wrapper query into an equivalent legacy query. The wrapper therefore implements the execution strategy for the query. The transformation must be achieved both *correctness* and *efficiency*. It is correct if the legacy query is transformed from mappings defined as a schema transformation sequence. The well-defined mapping from the wrapper schema to the physical schema makes the correctness issue. But producing an efficient execution strategy depends on the legacy query processing capabilities. A same wrapper query may lead to many execution strategy according to the query processing capabilities of the underlying legacy DMS. Since the query processing depends on the legacy system, there is as many strategies as legacy system families:

- The *capability of legacy DMS* may be different, which prevents uniform treatment of queries managed by wrappers;
- Similarly, the *cost of processing queries* may be different on different legacy DMS. This increases the complexity of comparing cost functions among wrappers;
- The *local optimization capability* of each legacy DMS may be quite different.

Query translation and result formation

Query translation is the translation of commands from one language to another. Different languages are used to manipulate data represented in different data models. Even when two DMS support the same data model, differences in their query language (e.g., QUEL and SQL) or different versions of SQL supported by two relational DMS can contribute to heterogeneity. Most of the existing wrapper prototypes provide some support for translation from the wrapper access language to the legacy access language. If the legacy DMS has more capabilities

than the wrapper interface the latter does not use extra features. However, if the wrapper is more expressive than the DMS, the translation processor must take into account this difference.

We recall that our approach is schema transformation oriented in that we focus on providing mechanisms for defining schema correspondence between the physical and wrapper schemas, and, on then using that equivalence to automatically perform the query mappings. Moreover, to make wrapper operational, wrapper queries must be translated from one language to another. The query transformation process can then be expressed as follows: *translate queries between two schemas and two languages by using language and schema mappings*.

We present an informal description of the query transformation process. Figure 5-7 shows the process. To separate the schema and query mappings, we introduce an internal form of the query, which is independent of the syntax of any query. In fact, during the translation process, the wrapper query Q_1 is first stripped off, creating a canonical form Q_2 that captures purely the semantics of the query. Next Q_3 is formed by applying the schema transformation sequence on the constructs of Q_2 . Finally, Q_3 is translated into a query Q_4 understandable by the legacy DMS.

We can now state the three main successive steps of query translation:

- *Language mappings*: syntactic translation of the wrapper query into an internal form;
- *Schema mappings*: semantics translation of the query using the schema transformation sequence that defines the mappings between the physical and wrapper schemas;
- *Language mappings and optimization*: syntactic translation of an internal form into a query based on the DMS query language;

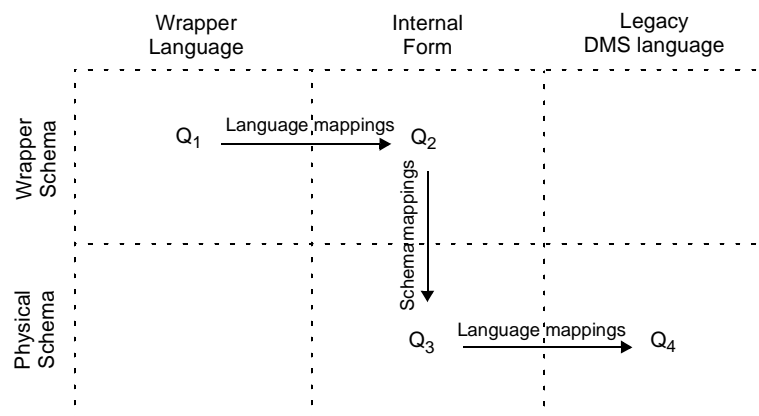


Figure 5-7: Language and schema mappings of a wrapper query Q_1 into a legacy DMS query Q_4 .

Example

To illustrate the query transformation, we use the physical and wrapper schemas in Figure 5-8. For simplicity, we use the relational model as the wrapper data model. The differences in terminology structure can easily be observed in this figure. The mappings Σ are defined as a chain of schema transformations (ET-att and desaggregation of Address).

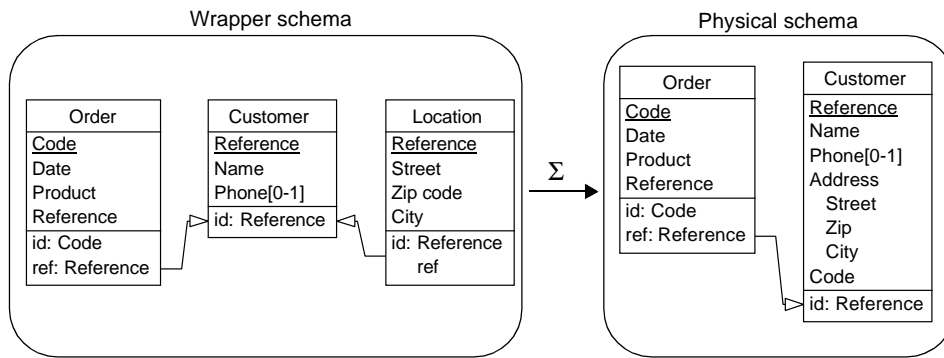


Figure 5-8: Query transformation example based on the schema transformation between the wrapper and physical schemas.

The following query (Q_1) expressed in

Q_1 : select C.name
 from ORDER O, CUSTOMER C, ADDRESS A
 where O.reference = C.reference
 and A.reference = C.reference
 and A.city = 'Namur'
 and O.date > '09-09-2000';

Step 1. Source language translation of the query: the input query is translated into an internal form. The query Q_1 is translated into the a conjunctive query [Chandra, 1977]. This translation provides a set expression by expanding any nested structures and nested queries.

Q_2 : {name | (\exists reference, city, date) (Order(reference, date) \wedge Customer(reference, name, city) \wedge Address(reference, city) \wedge (city='Namur') \wedge (date>'09-09-2000'))}

Step 2. Schema translation of the query: This translation is based on the transformations (considered the reverse way) used to produce the wrapper schema from the physical schema, such as ET-att and Desaggregation of Address, and a domain tranformation of

date.

Q_3 : {name | (\exists reference, add_city, date) (Order(reference, date) \wedge Customer(reference, name, add_city) \wedge (add_city='Namur') \wedge (date>'09:09:2000'))}

Step 3. Target language translation of the query: this last step in the translation process translates the conjunctive query into the legacy query language.

Q_4 : select C.name
 from ORDER O, CUSTOMER C,
 where O.reference = C.reference
 and C.add_city = 'Namur'
 and O.date > '09:09:2000';

Query optimization

An important aspect of query processing is query optimization. Because many execution strategies are correct transformations of the same wrapper query, the one that optimizes (minimizes) resource consumption should be retained. The query optimization must use the known legacy methods for data access. Therefore, the complexity of legacy access operations, which directly affects the wrapper execution time, dictates some principles useful to elaborate query plan strategy.

Optimization can be performed using two main techniques. *Heuristic rules* are used for ordering the operation is a query execution strategy. Heuristics are usually complemented with the use of a cost model which systematically evaluates the cost of different execution plans. For instance, a main heuristic rule states that SELECT and PROJECTION operations should be applied before the join and other binary operations. This is because SELECT and PROJECTION operations usually reduce the size of intermediate files.

As for query optimization using a cost model, the optimization first generates different execution strategies for a query and then systematically estimates and compares the costs of execution each execution strategy before selecting a strategy with the lowest cost.

Characterization of query processors. It is quite difficult to evaluate and compare query processors in the context of legacy systems because they may differ in many aspects. In what follows, we list important characteristics of query processors that can be used as a basis of comparison:

- Most works on query processing can be done by the legacy DMS itself because its access language gives the system many opportunities for optimization.
- A query may be optimized at different times relative to the actual time of query execution. Optimization can be done statically before executing the query or dynamically as the query is executed. Static query optimization is done at wrapper generation time.

A drawback of wrappers in data processing is due, in part, of the hiding the details about the physical constructs of the data. That doesn't allow the expression of optimized queries since the physical organization can't be exploited.

Example

Consider an application that issues a selection query to a wrapper. The underlying legacy information data it accesses are recorded in COBOL files. Assume the three queries of Figure 5-9 processed in the wrapper. Each query includes a different selection statement: the first one is made up of an attribute that is a COBOL record and access key; the second one, an attribute that is a COBOL alternate record and access key; and the last one, an attribute on which no access key has been implemented.

The wrapper processes the queries of Figure 5-9 by transforming it into a COBOL operations in taking the physical characteristics of the COBOL files into account. The simple look at the physical accesses suggests two principles. First, because primary access is the fastest access and only returns one instance, it should be considered as the priority access. For instance, in the first query of the Figure 5-9, because Code is defined as a recorded key in the COBOL files, the wrapper should process an indexed access. Second, operations should be ordered by decreasing time so that sequential access can be avoided or delayed.

Wrapper query	Constraint	Access number
Select Address From Customer Where Code='HTB710'	code is a record key (and an access key)	1
Select Address From Customer Where Name='THIRAN'	name is an alternate record key with dupli- cates (an access key)	>=1
Select Address From Customer Where Phone like '081*'	none	>=1

Figure 5-9: Wrapper query and optimization.

5.4.5 Semantic Integrity Control

Another important and difficult problem for a wrapper is how to guarantee the *data consistency*. A database is said to be consistent if the database satisfies a set of constraints, called semantic integrity constraints. Maintaining a consistent database requires various mechanisms such as concurrence control, reliability, protection, and semantic integrity control. *Semantic integrity control* ensures database consistency by rejecting update programs which leads to inconsistent database states, or by activating specific actions on the database state, which compensates for the effects of the update programs. Note that update database must satisfy the set of integrity constraints.

Implicit constraint concept

As far as wrapper is concerned, two main types of integrity constraints can be distinguished:

explicit constraints and implicit constraints:

- *Explicit constraints* are properties of constructs that are declared through a specific DDL statement.
- *Implicit constraints* are properties that hold in constructs, but that have not been declared explicitly. Through analysis of the DDL statements alone, the implicit constraints remain undetected.

As a result, the physical schema holds only the explicit constraints of a database whereas the wrapper schema holds both its explicit and implicit constraints.

Example

The most popular example certainly is that of foreign key. Let us consider the following example, in which two tables, linked by a foreign key, are declared. We can say that this foreign key is an explicit construct, insofar as we have used a specific statement to declare it.

```
create table CUSTOMER(C-ID integer primary key,
                     C-DATA char 80)
create table ORDER(O-ID integer primary key,
                  OWNER integer
                  foreign key(OWNER) references CUSTOMER)
```

The following program represents a fragment of an application in which no foreign keys have been declared, but which strongly suggests that column OWNER should behave as a foreign key. If we are convinced that this behavior must be taken for an absolute rule, then OWNER is an implicit foreign key.

```
create table CUSTOMER(C-ID integer primary key,
                     C-DATA char 80)
create table ORDER(O-ID integer primary key,
                  OWNER integer)
...
exec SQL select count(*) in :ERR-NBR from ORDER
      where OWNER not in (select C-ID from CUSTOMER)
end SQL

if ERR-NBR > 0 then
  display 'Referential constraints :', ERR-NBR, ' violations';
```

Implicit constraint emulation

While the DMS manages the explicit constraints (i.e., constraints defined in the physical schema), the wrapper emulates the implicit constraints by rejecting updates that violate implicit constraints (see Figure 5-10). A major difficulty in designing an integrity subsystem is

finding efficient enforcement algorithms.

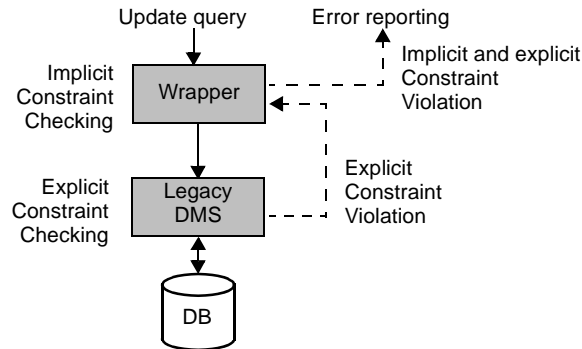


Figure 5-10: DMS and Wrapper management of explicit and implicit constraints.

Methods of inconsistent update rejections. Two basic methods permit the rejection of inconsistent updates. The first one is based on the *detection of inconsistencies*. The update is executed, consisting a change of the database state D to D_u . The enforcement algorithm verifies, by applying tests derived from these implicit constraints, that all relevant constraints hold in state D_u . If state D_u is inconsistent, the wrapper restore state D by undoing u . Since these tests are applied after having changed the database state, they are generally called *post-tests*. This approach may be inefficient of a large amount of work (the update of D) must be undone in the case of an integrity failure. The second method is based on the *prevention of inconsistencies*. An update is executed only if it changes the database state to a consistent state. The instances subject to the update are either directly available (in case of insert) or must be retrieved from the database (in the case of the deletion or modification). The enforcement algorithm verifies that all relevant constraints will hold after updating those instances. This is generally done by applying to those instances test that are derived from the integrity constraints. Given that these tests are applied before the database state is changed, they are generally called *pre-tests*.

Wrapper and inconsistency prevention. To handle general assertions, pre-tests can be defined at the generation time. For simplicity, the method is restricted to updates that insert or delete a single instance of a single entity type. This method is based on the production, at generation time, of *implicit constraint checking* which are used subsequently to prevent the introduction of inconsistencies in the database. The definition of implicit constraint checking is based on the notion of implicit constraint that is emulated by a wrapper. An implicit constraint checking is a triple $\langle ET, T, C \rangle$ in which ET is an entity type of the wrapper schema; T is an update type; and C is an implicit constraint assertion ranging over the entity type ET in an update of type T . When an implicit constraint I is defined, a set of implicit constraint checking may be produced for the entity types used by I . Whenever an entity of ET involved in I is updated, the implicit constraint checking assertions that must be checked to enforce I are only

those defined on I for the update type.

Implicit constraint checking assertions are obtained by applying transformation rules to the wrapper schema. These rules are mainly based on the update and implicit constraint types.

Example

Consider the wrapper schema in the Figure 5-11. This schemas is made up of two entity types A and B. The implicit constraint checking assertions associated with the reference constraint are:

$\langle A, \text{INSERT}, C1 \rangle$ and $\langle B, \text{DELETE}, C2 \rangle$

where $C1$ is $\forall \text{NEW} \in \{\text{new instances of A}\}, \exists b \in \{\text{instances of B}\} : \text{NEW.A2} = b.B1$;

and $C2$ is $\forall a \in \{\text{instances of A}\}, \forall \text{OLD} \in \{\text{deleted instances of B}\} : a.A2 \neq \text{OLD.B1}$.

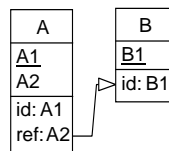


Figure 5-11: Wrapper schema example that illustrates a reference constraint.

Let us now illustrate the wrapper algorithm. We recall that a wrapper only emulates the implicit constraints. The algorithm acts in two steps. The first step verifies all the implicit constraints associated to each ET by implementing its constraint checking assertions. The second step consists of performing the update itself if no implicit constraint is violated.

Example

Let us consider the example of the implicit reference constraint of Figure 5-11. The main constraint checking assertions associated with this constraint are summarized in Figure 5-12. Both the predicative (SQL) and the procedural (pseudo-code) versions are given.

Entity Type	Update Type	Procedural Pattern
A	INSERT	read-first B(B1=A.A2) if found then create A; end-if;
B	DELETE	read-first A(A2=B.B1) if no found then delete B; end-if;
Entity Type	Update Type	SQL-like Expressions
A	INSERT	if exists (select * from B where B.B1= A.A2) then insert into A values (....)
B	DELETE	if not exists (select * from A where A.A2=B.B1) then delete from B where ...

Figure 5-12: The constraint checking assertions associated with a reference constraint.

Discussions

The main problem in supporting integrity control is that the cost of checking assertions can be prohibitive. Enforcing implicit constraint assertions is costly because it generally requires access to a large amount of data which is not involved in the database updates. Moreover, the problem is more difficult when implicit constraints are not supported by the legacy DMS.

Integrity control can be very complex if several checking assertions are defined for a same couple <ET, T>. The main problem is to decide the order of checking assertion enforcements. The critical parameter to be considered is their costs. That is, the order depends, among others, on the classes of the checking assertions and the amount of data access they involved.

5.5 InterDB Prototypes

The InterDB wrapper is in charge of managing the physical/object conversion of each local database. It offers remote Java objects to the client applications. InterDB wrappers and applications communicate over the Internet using the RMI communication protocols. The InterDB wrapper comprises two components, namely the logical wrapper and the object wrapper (Figure 5-13):

- The *logical wrapper* offers a SQL-like interface to an underlying database. The logical wrapper is a software layer that offers (1) an abstract interface based on the wrapper logical schema (WLS) of a legacy database; (2) the *Logical Query Language (LQL)*, a variant of the SQL language which uses naming convention and terminology defined in the wrapper logical schema. The logical wrapper is in charge of managing the logical/physical conversion of a local database. In addition, it makes the implicit constructs and constraints explicitly available.

- The *object wrapper* provides a remote object-oriented view of a logical wrapper. Such a wrapper is developed on top of a logical wrapper to give it a Java object-oriented interface based on the wrapper object-oriented schema (WOS) of the underlying database.

For performance reason, we have decided to develop the wrappers as program components dedicated to a local database. In particular, the logical/physical and object/logical mapping rules are hardcoded in the modules rather than interpreted from mapping tables.

With this architecture, a legacy database can be accessed in two ways (Figure 5-13): (1) through the logical wrapper interface or (2) through the object wrapper interface.

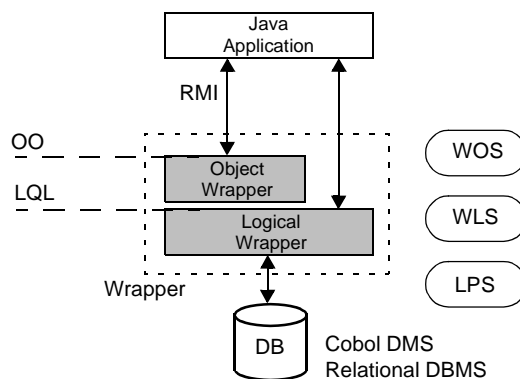


Figure 5-13: Logical and object wrappers in a Java environment.

Java technology

Java technologies (i.e., Java, Java-RMI, JavaBeans, JDBC, Java Naming, Object Serialization) are based on a distributed object-based client server model. These technologies bring several sophisticated capabilities to the development of wrappers. Java is a platform independent object-oriented programming language. It was designed to support applications on networks by bridging network and operating systems boundaries. In that way, Java applications can be run anywhere, provided the Java virtual machine is available. The Java compiler does this by generating the bytecode, which is independent of computer architectures. This bytecode is interpreted by the run time system.

The *Java RMI* provides remote method invocations on objects across Java virtual machines. It offers ORB-like functionality with the Java object model in the sense that it uses Java as both an interface definition language and an implementation language.

The *Java Native Interface* (JNI) enables the integration of code written in the Java programming language with code written in C++ [Liang, 1999]. JNI allows a program (e.g., the object wrapper) that runs within the Java Virtual Machine to operate with the logical wrappers that are written in legacy languages (C++ or Cobol).

5.5.1 Logical Wrapper

The logical wrapper comprises two components, namely the *logical module* and the *logical middleware* made up of the *logical server* and the *InterDB driver*, both dedicated to a database (Figure 5-14). The logical module offers a unique interface to the applications whereas the logical middleware provides a transparent distribution across the network.

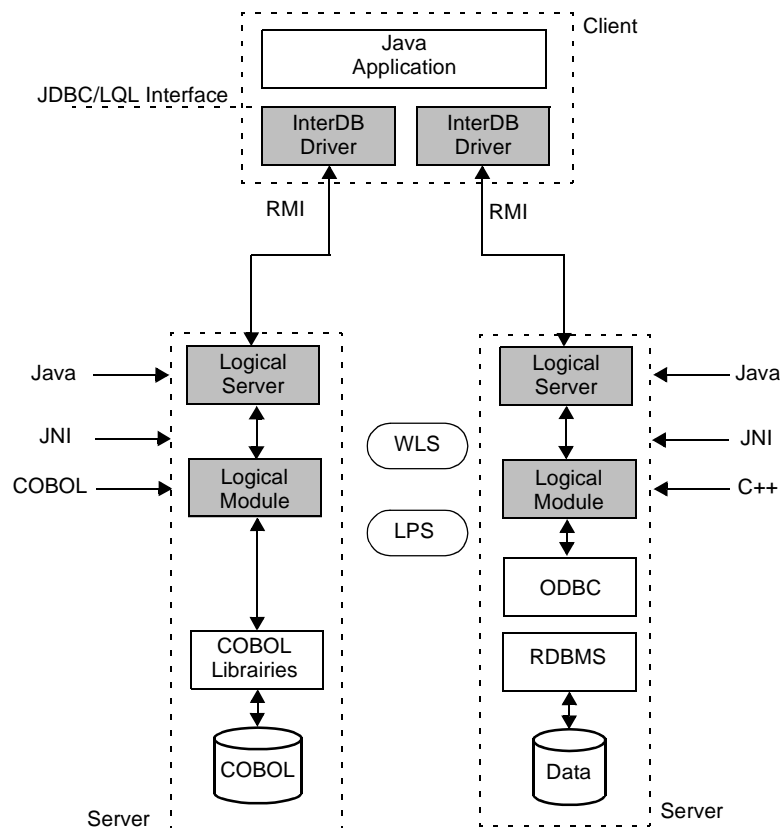


Figure 5-14: InterDB components of logical wrappers dedicated to COBOL files and relational databases.

Middleware

The middleware is made up of the logical server and the InterDB driver, both dedicated to a database.

Logical server. The logical server manages the communications between the logical wrapper and the InterDB driver. It offers the object distribution across the network: it receives the que-

ries sent by the InterDB driver and sends the result objects from the logical wrapper. We have used the RMI system to manage the communications.

InterDB driver. The JDBC-like driver is a Java API for executing LQL statements. Connected to the logical server, it provides to a Java application the JDBC-compliant classes and interface. The driver offers a standard API for heterogeneous databases based on a pure Java API and a common query language.

Logical module

All logical modules provide a common interface that is based on the wrapper logical model. Three of the major characteristics of the SQL-like language are that:

- It directly supports all of the concepts of the wrapper logical schema.
- It includes selection and update queries.
- Its query mappings are hardcoded and generated from the schema transformations between its physical and wrapper schemas (see Chapter 6).

We have built hardcoded logical modules for relational databases (Oracle, InterBase and Access) and others for files managers (COBOL programs). The modules for relation databases have been written in C++ using ODBC for accessing the legacy databases whereas the modules for files managers have been entirely written in COBOL (Figure 5-14). This implementation strategy has been motivated by the fact that the languages and the capacities of these DMS are different. Figure 5-15 draws up the services the logical modules provide and compares their complexity according to the underlying DMS.

Wrapper Services	Cobol	Relational DMS
Syntactic analysis	yes	yes
Query translation	yes (complex: query and language translation)	yes (easy: only query translation)
Cursor concept	yes	no (delegated to ODBC)
Access optimization and processing	yes	no (delegated to DMS)
Error reporting	yes	yes
Semantic Integrity Control	yes (implicit foreign keys and identifiers)	yes (implicit foreign keys and identifiers)
Memoryless	yes	no
Authorization control	yes	no (delegated to DMS)
Transaction management	no	no (but can be delegated to DMS)
Failure management	no	no

Figure 5-15: Wrapper services implemented by the InterDB prototypes, according to the underlying DMS.

ODBC technology

The ODBC interface [Geiger, 1995] allows the logical wrapper to access data from a wide range of RDBMS. Each logical wrapper uses the same code to communicate with a relational database through any RDBMS. Therefore, the logical wrapper is independent of a particular relational RDBMS.

Client application example

The client application example covers the basic use of the JDBC-like interface of logical wrappers. It shows a Java client application that accesses to a logical wrapper that offers the wrapper logical schema of the Figure 5-16. The Java client application illustrates the steps to connect, query and print the results of a wrapper request. The query retrieves all the instances of the entity type Customer for which Ncli=1. Figure 5-17 shows the Java code of the client application; Figure 5-18 shows the Java definition of the entity type Customer.

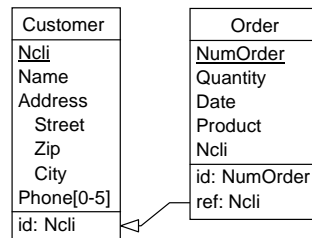


Figure 5-16: Wrapper logical example.

```
import java.util.Vector;
class test
{
    public static void main (String args[])
    {
        try
        {
            Connection con = DriverManager.getConnection("fuligule.info.fundp.ac.be/ServerLQL",
                                                         "SYSDBA", "masterkey");
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * FROM CUSTOMER WHERE NCLI=01;");
            CUSTOMER ca;
            if (rs.first())
            {
                System.err.println("\nCustomer:");
                ca = (CUSTOMER)rs.getObject("1");
                System.err.println("customer:nom:"+ca.NAME);
                System.err.println("customer.address.city:"+ca.ADDRESS.CITY);
            }
        }
        catch (LQLErrorException e) {System.err.println("LQL Error:"+e);}
    }
}
```

Figure 5-17: Java application example accessing to the logical wrapper through the InterDB driver.

```
import java.io.*; // Serializable since these are distributed objects
import java.util.Vector;
public class Customer implements Serializable {
    public Integer custID;
    public String name;
    public oaddress address;
    public Date birth-date;
    public Vector phone;
}

import java.io.*;
public class oaddress implements Serializable {
    public Integer number;
    public String street;
    public Integer zip;
```

```

    public String city;
}

```

Figure 5-18: Java definition of the entity type Customer.

5.5.2 Object Wrapper

The *object wrapper* provides a remote object-oriented view of a local database. It is a Java-written server that provides a remote read-only object interface. This interface is made up of the objects defined in the wrapper object-oriented schema (WOS) of the underlying database. We recall that the wrapper object-oriented schema is obtained by applying a model translation of the wrapper logical schema of the underlying database (Chapters 3 and 4).

The object wrapper doesn't support any query language. It only provides an object-oriented framework (i.e., object types and methods) for manipulating read-only data.

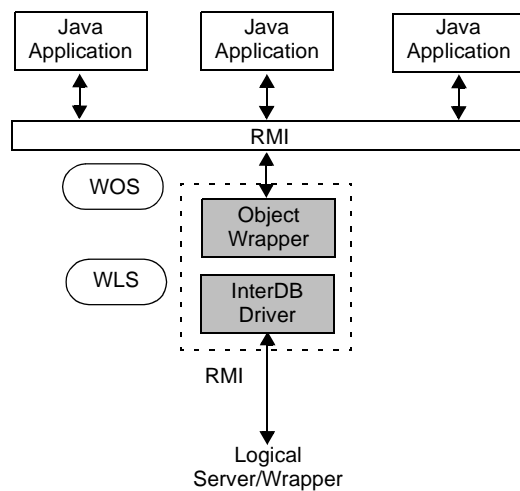


Figure 5-19: Object wrapper implementation and architecture.

The communication architecture of the object wrapper is shown in Figure 5-19. On one hand, the object wrapper uses the InterDB driver to access to the logical wrapper of an underlying database. On the other hand, it communicates with the Java application client by using the RMI communication protocols.

Client application example

To illustrate the remote object interface of the object wrapper, we use the wrapper object schema of Figure 5-20 that is the object-oriented view of the wrapper logical schema of Figure 5-16. Each object type corresponds to a Java remote object that can be manipulated by any Java client applications by using the RMI protocols. An example of the Java remote ob-

ject definition of Order is shown in Figure 5-21.

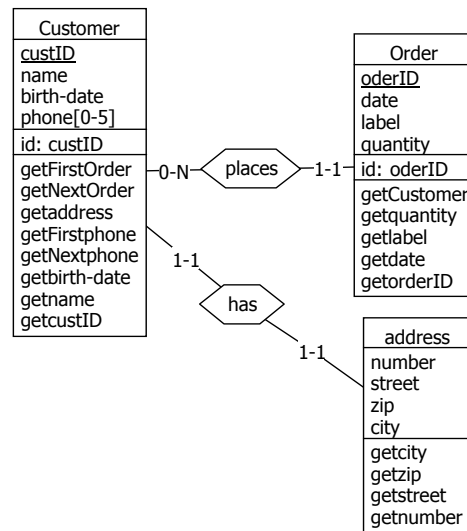


Figure 5-20: Wrapper object-oriented schema example.

```

import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface Order extends Remote
{
    Customer getCustomer() throws RemoteException, DOException;
    Customer getNextCustomer() throws RemoteException, DOException;
    Integer getquantity() throws RemoteException, DOException;
    String getlabel() throws RemoteException, DOException;
    Date getdate() throws RemoteException, DOException;
    Integer getorderID() throws RemoteException, DOException;
} //End interface
  
```

Figure 5-21: Remote object interface example: the object type Order.

Wrapper Development

In which the development of a methodology for developing wrappers in a semi-automatic way is discussed. Our experience in building wrappers is presented and some issues in wrapper generation are discussed. This leads to a proposal of wrapper development based on a practical and generic approach.

6.1 Introduction

This chapter deals with developing wrappers for legacy data systems. Developing wrappers, as we will see through this chapter, is an extensive and complex engineering activity that involves experiment and method. This chapter describes the InterDB development of wrappers and summarizes our experience in building data wrappers with Cobol files and relational databases.

Our approach of wrapper development addresses the challenge of DMS diversity by proposing a generic methodology for the wrapper development while taking a schema transformation approach to mapping definition. The key features of our approach of the wrapper development can be summarized as follows:

- *Hardcoded wrapper:* the wrappers are developed as program components dedicated to a database. The mapping rules are therefore hardcoded in the wrappers rather than interpreted from mapping tables.
- *Generated wrapper:* the wrappers are generated from mapping and schema specification.
- *Schema transformation-based wrapper generation:* the mapping rules are defined as schema transformations that are used to automatically generate the query mappings.

The approach presented in this chapter attempts to answer the wrapper development systematically and provides a road map to proceed with this venture. Our objective is to provide a checklist of issues and suggested courses of actions which can be modified, extended, and customized depending on the specific needs.

6.2 Wrapper Development Baselines

6.2.1 Observations

We have built hard-coded wrappers for several legacy data systems, including relational databases and COBOL files. Wrappers that are more than 10,000 LOC long are not uncommon, so that developing them represents an important effort in extending, integrating and reusing legacy systems. We observed that only a small part of the code of these wrappers actually deals with a specific data source. The other part is common to all the wrappers of a DMS family. We also demonstrated that the code specific to a particular legacy data system performs the structural and instance mappings and that these mappings can be modeled through semantics-preserving transformations. Therefore, it is possible to produce the procedural code of the specific wrapper automatically and to build a common generator for all the wrappers of a family of DMS.

However, based on the experiences in our application project areas of city administration systems, we have also stated that the formalized mappings cannot cover all the complexity of a data source (for instance, conflicts occurring among data inside the legacy system itself). So, we must admit that a part of the wrapper code have to be built manually. This is acceptable if the manual intervention points are clearly identified in the wrapper structure.

6.2.2 Wrapper Dimensions

Based on these observations, we define three dimensions of a wrapper dedicated to legacy data sources (Figure 6-1): (1) the *model wrapper*; (2) the *instance wrapper*; and (3) the *upper wrapper*. The first two dimensions are automated whereas the third dimension is built manually. The challenge is to reduce as much as possible the manual part.

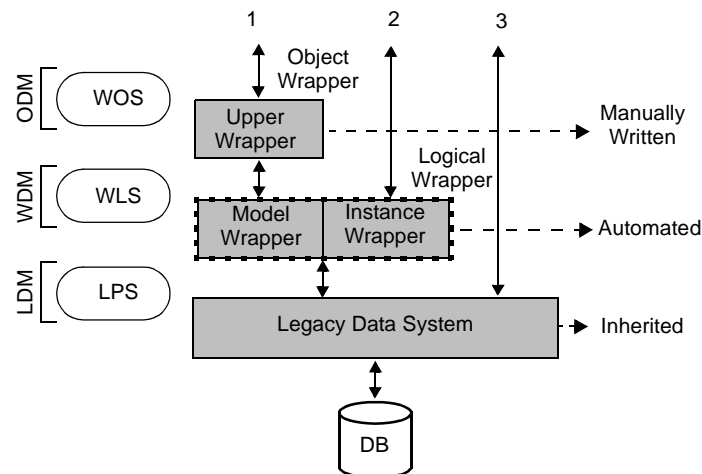


Figure 6-1: The three dimensions and schemas of a wrapper and the three ways for accessing a legacy data system.

With such an architecture, a legacy data system can be accessed in three ways (Figure 6-1): (1) through the upper wrapper interface; (2) through the interface of the basic wrapper; or (3) through its legacy interface.

Model and instance wrappers. The model wrapper is based on a legacy DMS family whereas the instance wrapper operates within a particular legacy data system. These two components form the *logical wrapper*. The logical wrapper wraps the legacy data system and offers an interface based on the logical schema of the wrapped data system. The logical wrapper converts data and queries from the legacy data model (LDM) to the logical wrapper model (WDM). The logical wrapper relies on schemas descriptions and mappings to translate queries and to form the result instances. That is, they can be complex if the mapping rules are complex and the wrapper data model is rich. As a result, a realistic logical wrapper should be based on an operational model, such as the wrapper logical model described in Chapter 3, and a realistic set of mapping rules.

The model wrapper is made up of the code common to wrappers dedicated to a DMS family, and can be written once for all. The instance wrapper is a program component dedicated to a particular database. It is based on the formalized physical/logical mapping rules. As we will see, it can be automatically generated from schema and mapping description.

The logical wrapper can be instantiated in two levels (Figure 6-2):

- *at the DMS level:* the model wrapper and the generator of the instance wrapper;
- *at the level of a particular legacy database:* the generated instance wrapper.

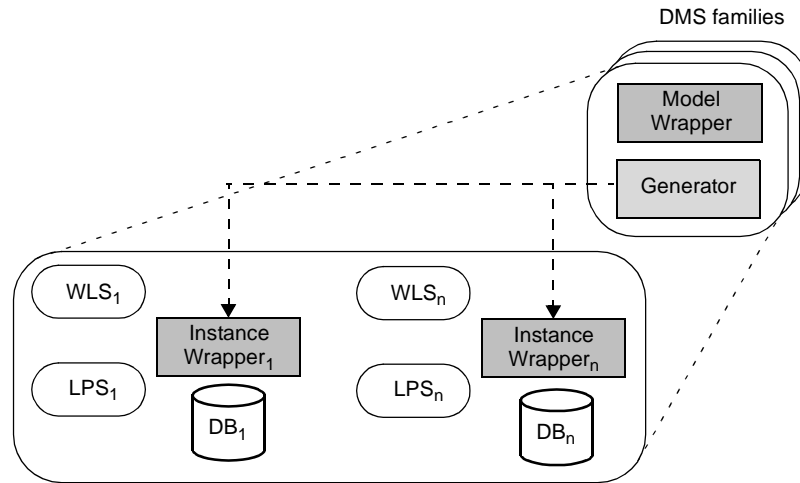


Figure 6-2: Logical wrappers: model wrappers and instance wrapper generators dedicated to a DMS family.

Upper wrapper. The upper wrapper is built on top of these two components. It offers an enriched view of the component schema. This view is based on a object-oriented model (ODM) that is highly generic and more flexible than the wrapper and legacy data models (see Chapter 3). The complex mapping rules (that cannot be taken into account by the logical wrapper) are programmed manually. The upper wrapper is named the *object wrapper* in the InterDB component architecture.

6.2.3 Wrapper Models and Schemas

The wrapper models include the *legacy data models* supported by the legacy databases and the two *canonical data models*, namely, the wrapper logical model and the object-oriented model:

- The *legacy data models* (LDM) are used to describe schemas compliant with DMS models, such as relational, CODASYL or IMS schemas.
- The *wrapper logical model* (WLM) is intended to describe all the structures and constraints that exist explicitly in all the legacy data models since a logical wrapper must be able to keep all the structures and constraints of any underlying legacy data schema based on any data model.
- The *object-oriented data model* (ODM) is defined highly expressive and more flexible than the legacy data models. Such a model generally includes structures and constraints (Δ) that are unknown in the wrapper data model and hence in the legacy data models.

All these models have been defined in Chapter 3. We recall that they are defined as specializations of the generic model. As far as wrapper development is concerned, we state the semantic relationships between these models:

$$WLM = \bigcup_i (LDM_i) \quad (\text{instance wrapper})$$

$$ODM = WLM + \Delta \quad (\text{upper wrapper})$$

$$ODM = \bigcup_i (LDM_i) + \Delta \quad (\text{whole wrapper})$$

Based on these relationships, we can also state the relationships in the schema level:

$$WLS = LPS + V \quad (\text{instance wrapper})$$

$$WOS = WLS + V' \quad (\text{upper wrapper})$$

$$WOS = WLS + V + V' \quad (\text{whole wrapper})$$

where V is the extra semantics of WLS emulated by the instance wrapper and V' is the extra semantics of WOS implemented in the upper wrapper.

6.2.4 Wrapper Mapping

Wrapper mappings are defined as schema transformations (see Chapter 4). A set of generic transformations Γ can be defined for the generic model. These transformations can be instantiated in order to get transformations defined in a schema level. The production of target schema S' from source schemas S , defined in non necessarily distinct submodels, can therefore be described as a subsequence of transformations from Γ . In particular, LPS-to-WLS and WLS-to-WOS can be defined by sequences of transformations.

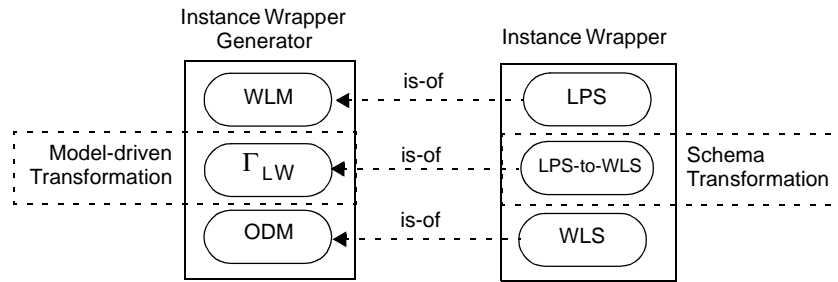


Figure 6-3: Instance wrapper generator and instance wrapper definition.

We can now refine the definition of the *instance wrapper generator*. It is based on a pre-defined set of schema transformations Γ_{LW} defined from two models (i.e., the legacy model and the wrapper model):

$\Gamma_{LW} \subseteq \Gamma$ and $\forall T \in \text{LPS-to-LWS}: T$ is defined in Γ_{LW}

Therefore, for a given generator, the set of transformations Γ_{UW} managed by the upper wrapper should be:

$\Gamma_{UW} : \Gamma_{UW} \cup \Gamma_{LW} = \Gamma$ and $\forall T \in \text{LWS-to-WOS}: T$ is defined in Γ_{UW}

Figure 6-3 shows the relationships between the instance wrapper and its generator in terms of models, schemas and mappings.

6.2.5 Logical Wrapper Architecture

The logical wrapper architecture shown in Figure 6-4 provides a generic wrapper framework, i.e., independent of a particular legacy database and of a DMS family. The dark grey boxes represent the model wrapper that is built once for a DMS family. A generator computes the light grey boxes for a particular legacy data system. They are built from the results of the reverse engineering process that will be discussed in Section 6.4.

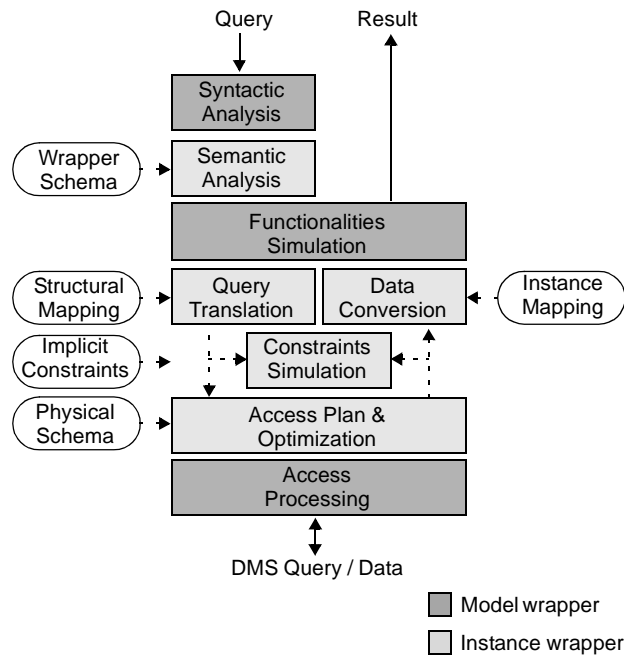


Figure 6-4: The components of the model and instance wrappers.

Model wrapper

The model wrapper is made up of components specific to a DMS family. For all the wrappers related to a DMS family, the model wrapper is written once. It includes the syntactic analysis, the functionality simulation and the access processing. As already stated in Chapter 5, the functionality simulation varies according to the expressive power of the DMS family.

Example

Let us compare the model wrappers for COBOL data sources and two relational DBMS: Oracle V5 and Oracle V8. Assume that the wrapper offers a SQL-like interface to the new applications. Assume also that the wrapper allows update queries and the two phase commit protocol. Figure 6-5 compares the complexity of the model wrappers according to the services they emulate and the underlying DMS.

Wrapper Services	Cobol	Oracle V5	Oracle V8
Syntactic analysis	yes	yes	yes
Query translation	yes (complex)	yes (easy)	yes (easy)
Cursor concept	yes	no (delegated to Oracle)	no (delegated to Oracle)
Transaction concept (pre-prepare-to-commit operator)	yes (complex)	yes (complex)	no (delegated to Oracle)
Access optimization and processing	yes	no (delegated to Oracle)	no (delegated to Oracle)

Figure 6-5: Model wrappers and the services they emulate according to the underlying DMS.

Instance wrapper

The instance wrapper is made up of components specific to a particular legacy database of a specific DMS family. This is the only dimension of a wrapper to be computed. The instance wrapper relies, among others, on:

- the structural mappings between the physical and wrapper logical schemas (LWS-to-LPS) for the **query translation** (for instance: the translation of the input query into SQL query or COBOL program codes);
- the instance mappings (lps-to-lws) that define the **data assembly** (for instance, building an object from a set of rows);
- the implicit constraints **V** to **simulate** (for instance, the simulation of a referential constraint);
- the structure of the data source (LPS) for the **access plan and optimization** (for instance, the presence of access keys or clusters).

Example

To illustrate the processes described above, let us consider a wrapper defined for COBOL files and that offers an SQL-like interface. Through the reverse engineering process (Section 6.4.1), we recover the physical schema and a semantically rich description of the COBOL files. During this process, the physical schema is enriched with an implicit foreign key (CustCode of Order). The names are translated to make them more meaningful. Finally, the physical schema is cleaned from its physical structures (access keys and files) (Figure 6-6).

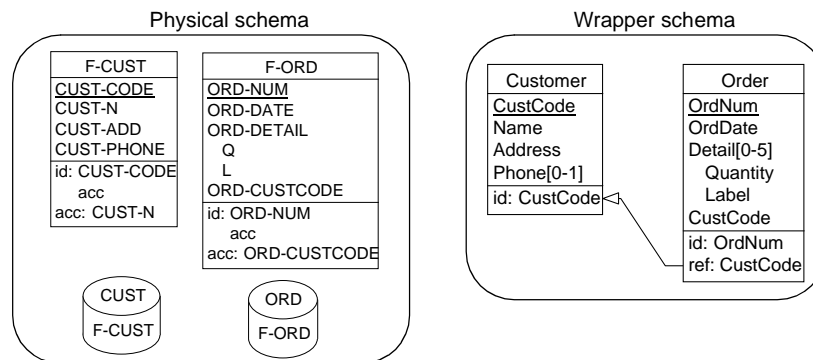


Figure 6-6: Physical and wrapper schema examples.

Let us focus on an instance wrapper dedicated to COBOL files and let us assume that the following update is processed by this wrapper: delete from customer c where c.CustCode = 'HTB710'. The main tasks of the wrapper are the following:

Query translation: translation of the query into COBOL program code:

```
DELETE CUST
* where CUST-Code = 'HTB710'
END-DELETE
```

Implicit constraints simulation: the foreign key CustCode of Order that references CustCode of Customer (with renaming):

```
* Before deleting:
MOVE CUST-Code OF F-CUST
  TO ORD-CustCode OF F-ORD.
READ ORD KEY IS ORD-CustCode.
IF FSTAT IS EQUAL TO "00"
  THEN MOVE 1 TO RECORD-FOUND
  ELSE MOVE 0 TO RECORD-FOUND
END-IF.
* If at least one record is found then no action
* else delete
```

Optimization: the deleting is positioned by CustCode that is an indexed record key (CUST-Code) in the physical schema, hence:

```
* Indexed access:  
READ CUST KEY IS CUST-Code  
DELETE CUST  
END-DELETE.
```

In short, by analyzing the input query, the wrapper dynamically defines a sequence of operations that correspond to the program code below:

```
MOVE CUST-Code OF F-CUST  
  TO ORD-CustCode OF F-ORD.  
READ ORD KEY IS ORD-CustCode.  
IF FSTAT IS EQUAL TO "00"  
  THEN MOVE 1 TO RECORD-FOUND  
  ELSE MOVE 0 TO RECORD-FOUND  
END-IF.  
IF RECORD-FOUND IS EQUAL TO 0  
THEN  
  READ CUST KEY IS CUST-Code  
  DELETE CUST  
  END-DELETE  
END-IF.
```

6.3 Methodology for Logical Wrapper Development

6.3.1 Overview

A large number of decisions need to be made to develop logical wrappers in a systematic way. We suggest a procedure which can be used to systematically understand the various issues, evaluate the difficulty, select an appropriate strategy, and implement/deploy a solution based on a strategy. This procedure concentrates on four key steps (Figure 6-7):

- wrapper definition;
- logical wrapper definition and generator implementation;
- wrapper packaging;
- generator application and maintenance.

The discussion in this section assumes that an initial stage has concluded that wrapping legacy database is the right strategy. Our goal here is to proceed further by determining and implementing the most appropriate approach for developing wrappers.

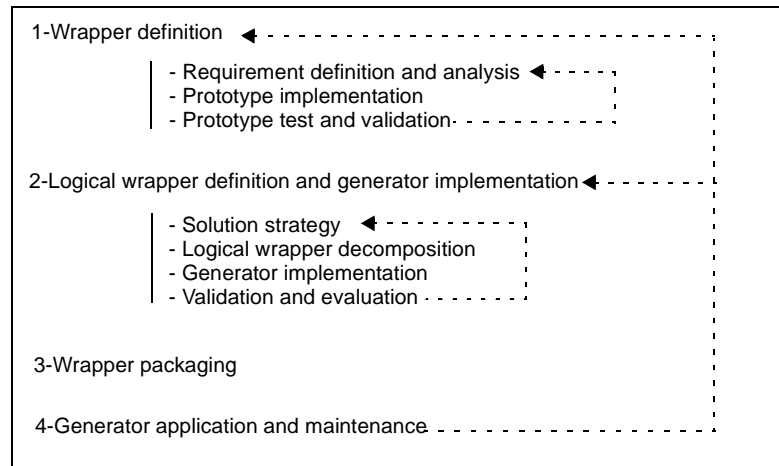


Figure 6-7: Main steps of the wrapper generator development with possible step backwards.

6.3.2 Development Baselines

Our experience in building wrappers demonstrates that their development must respect some baselines that make them operational and efficient. We summarize these baselines here before describing the methodology in details:

- *The start-up cost to write a wrapper should be small.* We require that the logical wrapper provides a set of basic services only. For instance, its data model, query processing capabilities and the set of mappings it manages must be operational and realistic.
- *The architecture should be flexible and allow for graceful growth.* A wrapper must be able to incrementally improved to offer more sophisticated services.
- *The architecture should be structured and allow reusing.* The common parts among wrappers must be easily identified and isolated from the parts specific to a particular wrapper.

6.3.3 Wrapper Definition

The wrapper definition is the first phase that lays the foundation for the wrapper development. Wrapper development is heavily dependent on the underlying DMS, the wrapper data model and the query language selected.

Requirement definition and analysis

The requirements gathered in this step are intended to drive the wrapper building. The spe-

cific activities are as following:

- Identify the legacy database(s) which need to be wrapped;
- Analyse the legacy databases according to their family. Determine and compare the capability of each DMS (e.g., query language, data model, functionality);
- Identify the client applications and/or tools which need to access the legacy databases through a wrapper. In particular, you need to know the specifications (e.g., wrapper model and query language) and services required by them (e.g., number of users, transaction management, etc.);
- Understand and document management and policy issues such as cost limitations, time considerations, security considerations, conformance to corporate standards and infrastructures;
- Determine the services (and their complexity) that must be managed/emulated by the wrapper for each DMS family. It results of the comparison between steps 2 and 3 in taken into account the requirements defined in step 4.

Prototype implementation

Implementation is concerned with building and deploying a wrapper prototype for each DMS family. Specifically, this activity includes detailed design and implementation of each services emulated by the wrappers. Implementation is largely dependent on the underlying DMS and its capability. Ideally, wrappers for the different DMS families should be implemented by using a common language. This promotes the reuseness of services along the different wrappers. However, a wrapper should use the legacy access techniques to improve data access performance. That is, it needs to access the native API of the underlying DMS by using the legacy language.

Well-designed prototypes are essential to the development of wrapper generators and should be designed to accomplish the following:

- Develop feedback from users;
- Use the experience to reduce time for developing a wrapper for another DMS family;
- Gain real insights into the infrastructure needed to support wrappers
- Develop an understanding of DMS families;
- Study the trade-offs in services among wrappers;
- Estimate the effort needed to add a new service to wrappers.

Prototype test and validation

Testing a wrapper is a challenging and expensive task. Why? Here are the main reasons. First, wrapper applications are complex and shaky because they employ several program languages and several technologies that must work together. Second, wrapper applications introduce many points of failures that require thorough testing. Third, the impact of the wrapper in the fonctionnement of the legacy components (middleware, networks, DMS, legacy applications)

must be thoroughly tested.

6.3.4 Logical Wrapper Definition

We require that the start-up automatic part of a wrapper is small. A wrapper provides a set of basic services only. Practically, we begin to manually write the whole wrapper as stated in Section 6.3.3. Then, according to a complexity and reusability criteria, we increase the automatic parts of the wrapper. We can also gradually add a service to a wrapper as more functionality is required later on.

Solution strategy

The solution strategy for each legacy DMS family and for each legacy database must answer the following key question: *is building a wrapper generator required?* (i.e., can the wrapper be easily written manually ?)

The wrapper building is generally a mixture of manually and automatic parts. Their exact proportion is based on the requirements above and an evaluation of the trade-offs among the two extreme approaches: a manually written wrapper versus an automated wrapper. The two key factors are (a) the number of legacy databases of a same DMS family; (b) the type and the complexity of services that the wrapper is to be simulated for a DMS family. The following guidelines can be used to make this decision:

- If the number of legacy databases which need to be wrapped is very high, then manually written wrapper isn't an appropriate approach;
- If a wrapper service is common of any legacy databases of a DMS family, then it can be written once and reused for all the wrappers of that DMS family;
- If a wrapper service is specific to a legacy database, then it can be either generated or written manually.

Logical wrapper decomposition

The purpose of this activity is to completely decompose logical wrappers into manually and automatic parts that will drive the wrapper generator buildings. We assume that the automatic parts are defined from schema transformations. The logical wrapper decomposition therefore involves, for each DMS family, the following two main tasks:

- Defining the set of transformation types supported by the instance wrapper and the prototype algorithm that emulates them;
- Defining the constraint integrities emulated by the instance wrapper and the prototype algorithm that emulates them.

Once each logical wrapper has been decomposed, a last task is to compare the common algorithms among the logical wrappers of different DMS families. The goal is to maximize the reusness of the wrapper components among them.

Generator implementation and validation

Generator implementation is concerned with generating wrapper code specific to schema transformation type and constraint integrity. For each schema transformation type emulated by the wrapper, this activity includes:

- Information extraction from history;
- Wrapper code generation;
- Test.

Wrapper code generation as well as wrapper test are largely dependent on the DMS family and the language used.

6.3.5 Wrapper Packaging

The wrapper author's final task is to package all the wrapper components as a complete wrapper development kit. For each DMS family, a wrapper development kit may include three kinds of components: the model wrapper that contains all components shared among all the wrappers of the DMS family; the instance wrapper generator; and the documentation that explains how to compile and link all the components of wrappers.

6.3.6 Generator Application and Maintenance

After the wrapper packaging has been completed, it can be deployed and supported and managed like any other technology. In particular, the following issues need to be considered:

- Estimate time and cost needed for wrapper deployment and support;
- Establish policies, procedures, and roles for wrapper deployment and support;
- Train analysts for wrapper development and support;
- Monitor for performance.

6.4 Methodology for Instance Wrapper Generation

After the logical wrapper has been defined and the instance wrapper generators have been implemented, the next step is to provide a methodology to prepare the generation. That is, the methodology aims to provide the schema and mapping specifications that are used for the instance wrapper generation. The main steps of the methodology are shown in Figure 6-8.

In this section, we develop a small example that illustrates some of the problems of instance wrapper generation. The example comprises two independent heterogeneous databases both describing aspects of a bookshop, that are required to be wrapped. The first one is made up of two COBOL files and the second one includes two relational tables

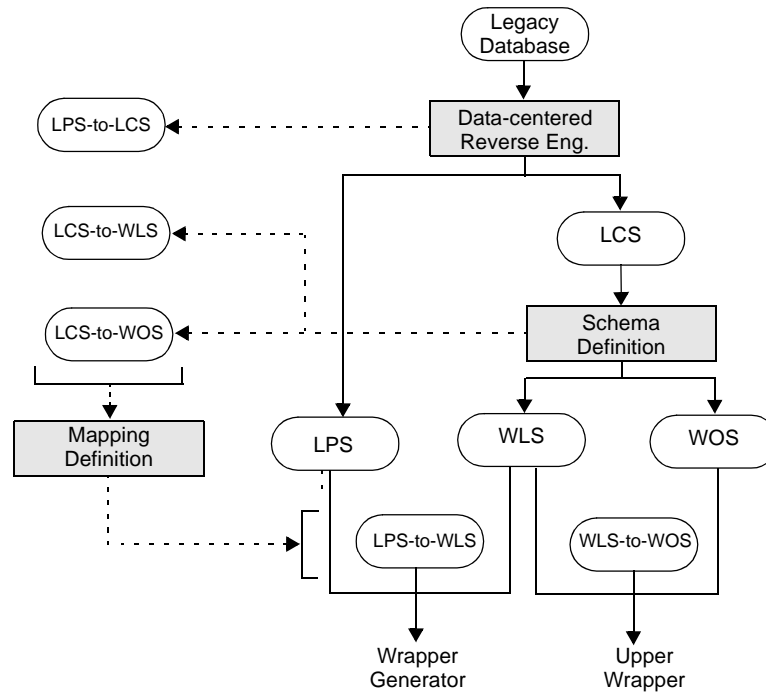


Figure 6-8: Main steps of the methodology for instance wrapper generation: (1) semantic recovery; (2) schema definition; (3) mapping definition.

6.4.1 Data-centered Reverse Engineering

Both the basic and object wrappers require to recover the physical, wrapper logical and object schemas of the legacy database. They also require to define the physical/logical and logical/object mappings modeled through compound semantics preserving transformations.

Extracting a semantically rich description from a database is the main goal of the *Data-centered Reverse Engineering process* (DRE). The InterDB approach relies on the general DBRE methodology that has been developed in the DB-MAIN project and the architecture of which is outlined in Figure 6-9. It shows clearly three main processes, namely the *physical extraction*, the *logical extraction* and the *DS conceptualization*. They will be described and illustrated in the following sections.

This methodology can be specialized according to the various data models which most legacy systems are based on, such as standard files, CODASYL, IMS and relational databases. The reader interested in more details on the DB-MAIN reverse engineering approach is suggested

to consult [Hainaut, 1996b].

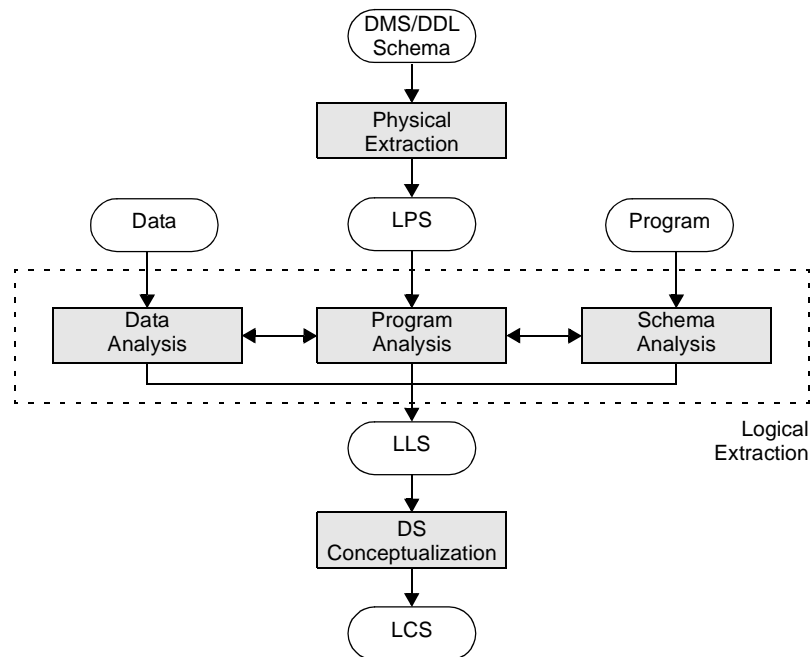


Figure 6-9: A Generic DRE methodology. The main processes extract the physical schema (LPS), refine it to produce the logical schema (LLS) and interpret the latter as a conceptual schema (LCS).

Physical schema recovery

This phase consists in recovering the physical schema (LPS) made up of all the structures and constraints explicitly declared. Databases systems generally provide a description of this schema (catalogue, data dictionary contents, DLL texts, file sections, etc.). The process consists in analyzing the data structure declaration statements (in the specific DDL) or the contents of these sources. It produces the physical schema (LPS) based on the data model (LDM) of the legacy database. The process is more complex for file systems, since the only formal descriptions available are declaration fragments spread throughout the application programs. This process is often easy to automate since it can be carried out by a simple parser which analyses the DMS-DDL texts, extracts the data structures and expresses them as the LPS. For instance, several popular CASE tools include some sorts of extractors, generally limited to RDB, but sometimes extended to COBOL files and IMS databases.

Example

By analyzing the COBOL programs and SQL DDL scripts, we can extract the local

physical schema of each database. Figure 6-10 shows the extracted schemas according to their data model.

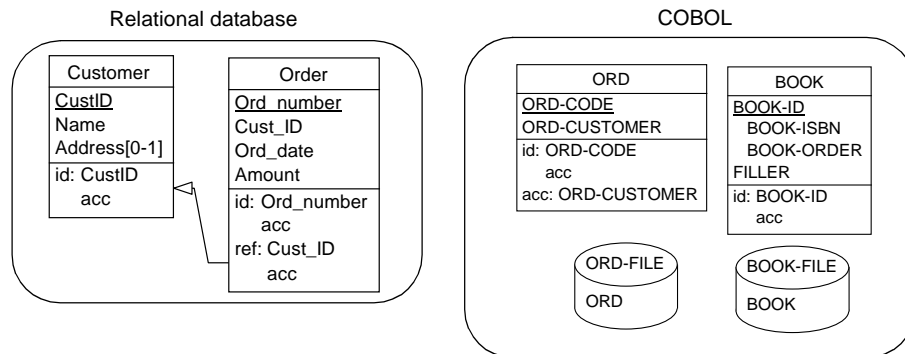


Figure 6-10: The local physical schemas. The relational database (left) comprises two tables, namely CUSTOMER and ORDER. The COBOL database (right) is made of two files and two record types (ORD and BOOK). BOOK-ID is a compound field and ORD-CUSTOMER is a non-unique alternate record key.

Schema refinement

The LPS is a rich starting point that must be refined through the analysis of the other components of the applications (views, subschemas, screen and report layouts, programs, fragments of documentation, program execution, data, etc.). This schema is then refined by specific analysis techniques that search non-declarative sources of information for evidences of implicit constructs and constraints. This schema is finally cleaned by removing its non-logical structures such as access keys and files. In this phase, three techniques are of particular importance.

- *Program analysis.* This process consists in analyzing parts of the application programs (the procedural sections, for instance) in order to detect evidences of additional data structures and integrity constraints.
- *Data analysis.* This refinement process examines the contents of the files and databases in order (1) to detect data structures and properties (e.g., to find unique fields or functional dependencies in a file), and (2) to test hypotheses (e.g., Could this field be a foreign key to this file?).
- *Schema analysis.* This process consists in eliciting implicit constructs (e.g., foreign keys) from structural evidence, in detecting and discarding non-logical structures (e.g., files and access keys), in translating names to make them more meaningful, and in restructuring some parts of the schema.

The end product of this phase is the *Local Logical Schema* (LLS) and the transformation sequences LPS-to-LLS.

Example

By applying the schema refinement, we obtain the local logical schemas of Figure 6-11, that make two hidden constraints explicit, namely a foreign key and a functional dependency in the COBOL database. They express the data structures in a form that is close to the DMS model, enriched with semantic constraints.

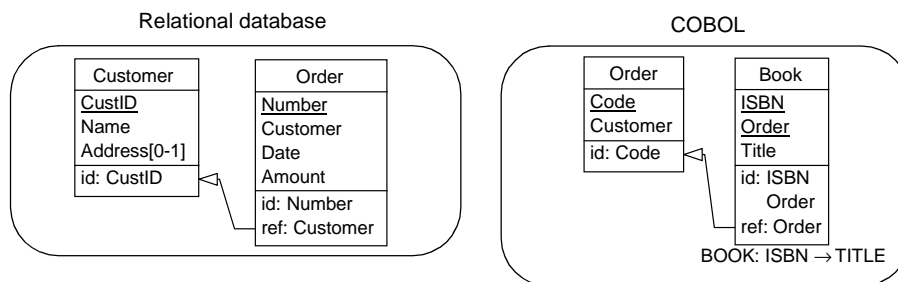


Figure 6-11: The local logical schemas of the relational database (left) and of the COBOL files (right). We observe the elicitation of an implicit foreign key and of a functional dependency in the COBOL database. The purely physical objects have been removed and names have been re-worked.

Schema conceptualization

This process addresses the semantic interpretation of a logical schema, from which one tries to extract a conceptual schema. The objective is to identify and to extract all the relevant semantic concepts underlying the logical schema. It mainly consists in detecting and transforming, or discarding, non-conceptual structures. Any logical schema can be obtained by a chain of transformations applied to the source conceptual schema. The conceptualization process can then be modeled as the undoing of the conceptual-to-logical translation, that is, applying the inverse transformations. Three different problems have to be solved through specific transformational techniques and reasoning.

1. *Untranslation.* Considering a target DMS model, each component of a conceptual schema can be translated into DMS-compliant constructs through a limited set of transformation rules. The identification of the traces of the application of these rules and the replacement of DMS constructs with the conceptual constructs they are intended to translate, form the basis of the untranslation process.
2. *De-optimization.* Most developers introduced, consciously or not, optimization constructs and transformations in their logical schemas. These practices can be classified into three families of techniques, namely structural redundancies (adding derivable constructs), unnormalization (merging data units linked through a one-to-many relationship) and restructuring (such as splitting and merging tables). The de-optimization process consists in identifying such patterns, and discarding them, either by removing or by

transforming them.

3. *Normalization.* This process is similar to the conceptual normalization process. It consists in restructuring the raw conceptual schema obtained in Steps 1 and 2 in order to give it such qualities as readability, conciseness, minimality, normality and conformity to a corporate methodology standard.

The result of this process is the local conceptual schema (LCS) and the transformation sequences LLS-to-LCS.

Example

The underlying semantics of the logical structures are extracted and give rise to the conceptual schema definition.

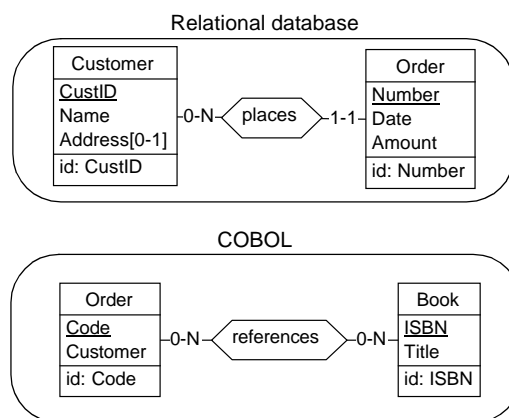


Figure 6-12: The local conceptual schemas (LCS) of the relational database (up) and of the COBOL files (down). The relational foreign keys have been transformed into relationship types. The BOOK record type has been normalized by splitting it into BOOK and REFERENCES, the latter being transformed into a many-to-many relationship type.

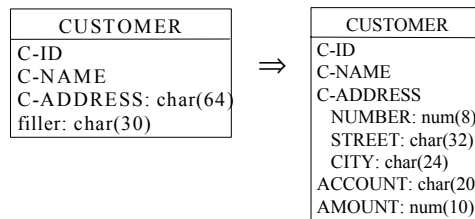
6.4.2 Catalog of Implicit Constraints and Constructs

The variety of implicit constructs can be fairly large, even in small systems. This section suggests just to mention the main implicit structures and constraints we found in reverse engineering processes [Hainaut, 1996]. We will briefly describe the most common problems we found when recovering the logical schemas of COBOL, SQL (ORACLE, InterBase).

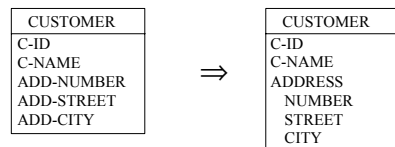
Implicit compound attribute

A field, or a full record type, declared as atomic, has an implicit decomposition, or is the con-

catenation of contiguous independent fields (C-ADDRESS, filler). This pattern is very common in standard files, but it has been found in modern databases as well, for instance in relational tables.

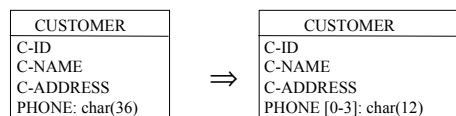


A sequence of seemingly independent fields (ADD-NUMBER, ADD-STREET, ADD-CITY) are originated from a source compound field which was decomposed. This is a typical situation in relational databases.



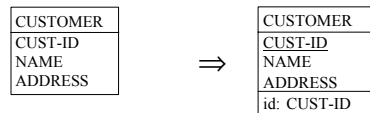
Implicit multivalued attribute

A field, declared as single-valued, appears as the concatenation of the values of a multivalued field (PHONE). Relational databases commonly include such constructs.



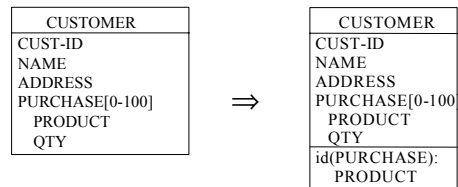
Implicit identifiers

The identifier (or unique key) of a record type is not always declared. Such is the case for sequential files for example. The fact that the CUST-ID values are unique among the CUSTOMER records must be proved.



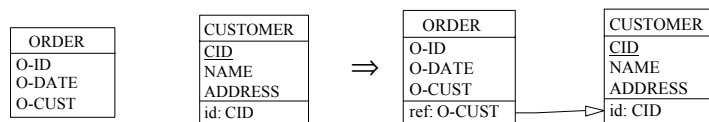
Implicit identifiers of multivalued attributes

Structured record types often include complex multivalued compound fields. Quite often too, these values have an implicit identifier. In each CUSTOMER record, there are no two PURCHASE values with the same PRODUCT value.



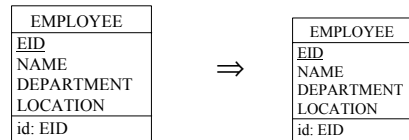
Implicit foreign keys

In multi-file applications, there can be inter-file links, represented by foreign keys, i.e. by fields whose values identify records in another file. For instance, field O-CUST in record type ORDER is used to designate a CUSTOMER record.



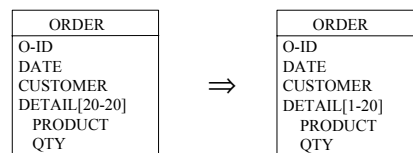
Implicit functional dependencies

As commonly recognized in the relational database domain, normalization is a recommended property. However, many actual databases include unnormalized structures, generally to get better performance. In the EMPLOYEE record type, the value of field LOCATION depends on DEPARTMENT, which a non-key field. This record type is in 2nd normal form only.



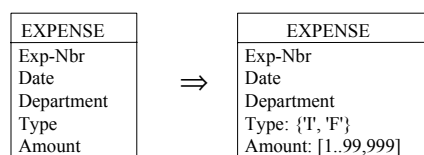
Finding exact minimum cardinality of fields and rel-types

Multivalued fields are generally declared as arrays, whose maximum size is specified by an integer, while the minimum size is not mentioned, and is under the responsibility of the programmer. For instance, field `DETAIL` has been declared as "occurs 20", and its cardinality has been interpreted as `[20-20]`. Further analysis has shown that this cardinality actually is `[1-20]`.



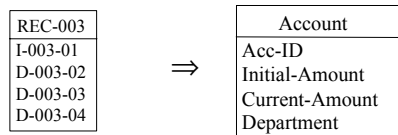
Implicit constraints on value domains

In most DBMS, declared data structures are very poor as far as their value domain is concerned. Quite often, though, strong restriction is enforced on the admissible values. In the example below, field `Type` has an enumerated domain, comprising two values "I" and "F", while values of field `Amount` must fall into the interval `[1..99,999]`.



Meaningful names

Some programming discipline, or technical constraints, impose the usage of meaningless names, or of very condensed names whose meaning is unclear. On the contrary, some applications have been developed with no discipline at all, leading to poor and contradictory naming conventions.



6.4.3 Wrapper Schema Definition

The objective is to identify and to extract all the constructs and constraints that are not supported by the wrapper models. This simply consists in a model translation of the LCS or LLS according to the abstraction level selected. The result of this process is the wrapper logical schemas and the object-oriented schemas (Figure 6-13).

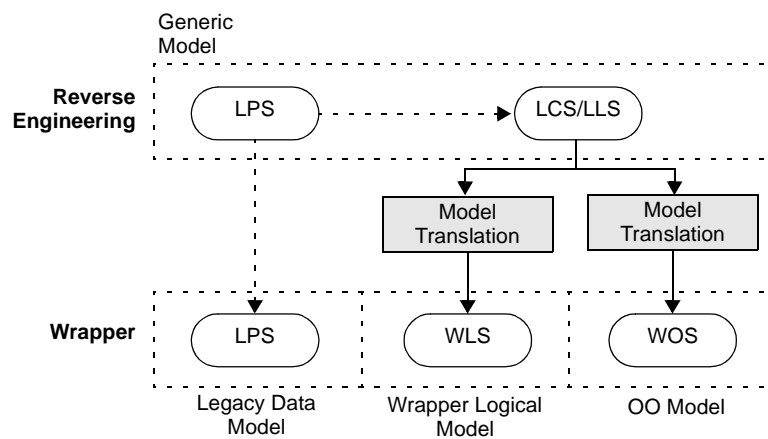


Figure 6-13: Wrapper schema definition: model translation of the physical and conceptual (or logical) schemas into the wrapper data models.

Example

To illustrate the wrapper schema definition, let us take up the example of Figure 6-11 and consider the LLS of the wrapper for COBOL files only. This schema is also depicted in the left side of Figure 6-14.

Let us suppose now that the wrapper model is the logical wrapper model defined in Chapter 3. Consequently, the model translation only consists in removing the functional dependency which is the only constraint not supported by the logical wrapper model. The result of this model translation is the wrapper schema WLS shown in the right part of Figure 6-14.

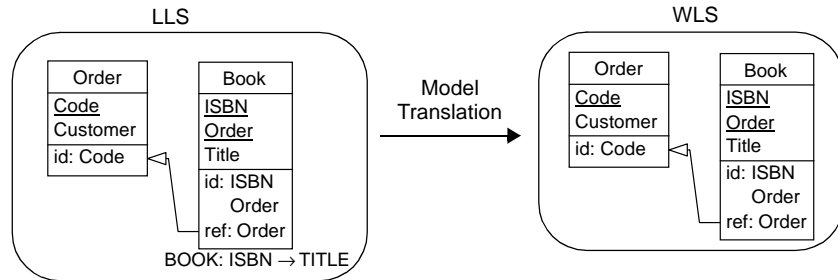


Figure 6-14: Wrapper schema definition example: LLS is converted into the wrapper logical model by dropping a constraint not supported by this model.

6.4.4 Mapping Definition

As shown in Chapter 4, the mappings are modeled through semantics-preserving transformations. Two transformation sequences have to be defined:

- the sequence emulated by the instance wrapper (LPS-to-WLS);
- the sequence not supported by the instance wrapper (WLS-to-WOS).

These sequences are built from the transformation sequences LPS-to-LCS, LCS-to-WLS and LCS-to-WOS got during the previous steps (Figure 6-15). That is:

$$\begin{aligned} \text{LPS-to-LCS} &= \text{Min}(\text{LPS-to-LLS} \circ \text{LLS-to-LCS}) \\ \text{LPS-to-WLS} &= \text{Min}(\text{LCS-to-WLS} \circ \text{LPS-to-LCS}) \\ \text{WLS-to-WOS} &= \text{Min}(\text{LCS-to-WOS} \circ \text{LPS-to-LCS}) \end{aligned}$$

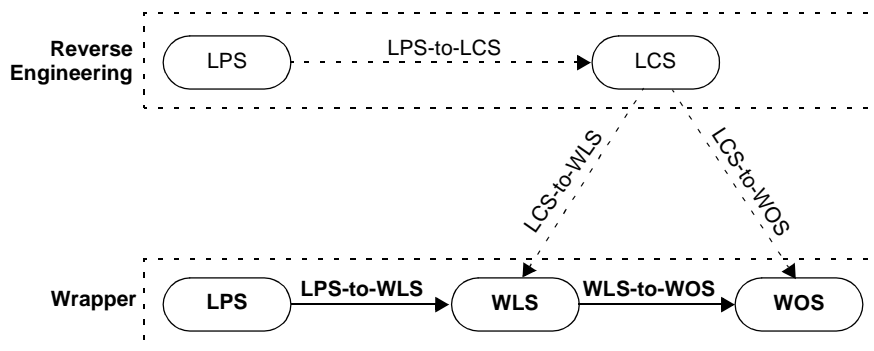


Figure 6-15: Wrapper mapping definitions: minimal sequences of compound transformations defined in the reverse engineering process.

6.5 Methodology for Upper Wrapper Development

The upper wrapper manages all the structures and constraints that are not defined in the wrapper model. It is built manually but can rely on some results of the reverse-engineering process: (1) the wrapper logical and the object schemas; and (2) the logical/object mappings (WLS-to-WOS).

The components of the upper wrapper can be allocated in several placements: they can be integrated in the logical wrapper; they can be developed independently and placed on top of the logical wrapper. Placing these components at the most appropriate place is an important issue that has yet to be worked out. Though we have no stable answer to this question yet, we can identify three major criteria of the placement of the upper wrapper components:

- *Optimization*: optimization of the whole wrapper performance;
- *Simplicity*: simplicity of the upper wrapper code;
- *Independence*: independence between the logical wrapper and the upper wrapper.

Wrapper Development Support

In which we analyse the main requirements that a CASE tool should meet for the development of wrapper. An operational CASE tool - DB-MAIN, which is intended to address some of these requirements, is then presented.

7.1 Introduction

Like any complex process, developing a wrapper cannot be successful without the support of adequate tools called *CASE tools*. Nevertheless, completely automating this process is unrealistic for real world systems. Hence the need for computer-based assistance tools which address several aspects of the development of wrappers (see Chapter 6).

Few tools are now available for building data wrappers for legacy information systems and database systems in particular (e.g., [Papakonstantinou, 1995] and [Vermeer, 1996]). Many of these tools, however, appear to be limited in scope, and are generally based on the quality and completeness of the database structures to be wrapped that cannot be relied on in many practical situations.

In such tools, it appears that only databases that can be processed are those that have been obtained by a rigorous database design method. This condition cannot be assumed for most large operational databases, particularly the oldest one. Moreover, these proposals are most often dedicated to one data model and do not attempt to elaborate techniques and reasonings common to several models, leaving the question of a general approach still unanswered.

7.2 CASE Tool Requirements

This section states some of the most important requirements an ideal CASE tool environment for the development of a wrapper should meet. Besides standard functions of a data-oriented CASE tool, an *ideal* CASE tool for the development of FIS should support the specific aspects of the development of FIS: (1) semantics recovery of the information sources, (2) mapping definition and (4) generating the wrapper procedural code. The requirements are induced by the analysis of each of these aspects.

7.2.1 General Support

Developing a wrapper is primarily a data-oriented engineering activity. Hence, the CASE tool must offer standard functions that are now provided by most CASE tools dedicated to data-oriented engineering. Moreover, since schema transformation is at the core of methodologies that manipulate schemas, the CASE tool must provide a rich set of transformation techniques.

Building a wrapper is basically an exploratory and often unstructured activity. Some important aspects of higher level specifications cannot be deterministically inferred. The tool must allow users to follow any working patterns, including unstructured ones. It should allow various engineering strategies; ranging from formal approaches to informal and pragmatic ones. In addition, the tool must be highly interactive.

There are no available tools that can satisfy all corporate needs in development of wrappers. In addition, current CASE tools already provide elaborated techniques that deal with some specific aspects of the design process. The CASE tool must communicate easily with the other development tools, exchanging specifications through common formats (such as XML, or a common repository).

7.2.2 Support of the Data-centered Reverse Engineering

Wrapper architectures require to support a great variety of legacy systems running on different platforms. These legacy systems include not only structured information sources but also semi-structured and even unstructured information [Conrad, 1999]. Customizable DBRE functions for automatic, interactive and assisted specification extraction should be available for each source types. They should be easy to customize and to program.

Moreover, the semantic enrichment requires a great variety of information: data structure, data, CASE repository, documentation, domain knowledge, etc. Several ways of viewing and querying these sources must be provided.

The canonical data model is designed to express all the semantics of the physical schema. It must be highly generic and more flexible than the legacy data models (see Chapter 3).

7.2.3 Support of the Mapping Definition

Developing a wrapper includes at least three sets of specification: the physical schema, the wrapper schema and the mappings between them. The forward and backward mappings between the schemas specification must be precisely and formally recorded in the repository of the CASE tool.

7.3 DB-MAIN

The DB-MAIN CASE environment [Hick, 2002] is a graphical, repository-based, software engineering environment dedicated to database applications engineering.

As far as wrapper development support is concerned, the DB-MAIN CASE tools and its InterDB extensions have been designed to address as much as possible the requirements developed in the previous section.

As a large-scope CASE tool, DB-MAIN includes usual functions needed in data analysis and design, e.g. entry, browsing, management, validation, transformation, as well as code and report generation. However, the rest of this chapter concentrates only on the main aspects and components of the tool which are directly related to wrapper development activities. In particular, we present the InterDB tools that have been built on top of DB-MAIN by adding concepts and processors that support the wrapper generation.

7.3.1 User Interface

User interaction uses a fairly standard GUI. Browsing through several sources require an adequate presentation of specifications. It appears that more than one way of viewing them is necessary. For instance, a graphical representation of schemas allows an easy detection of certain structural patterns, but it is useless to analyze the attribute domains. DB-MAIN currently offers six ways of presenting a schema (four hypertext views and two graphical views). Four screens of them are illustrated in Figure 7-1.

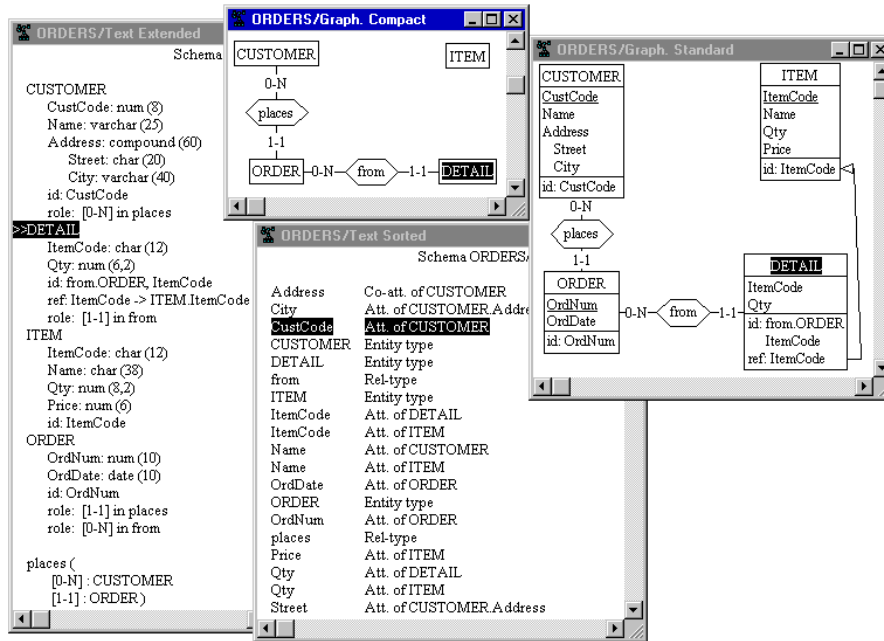


Figure 7-1: DB-MAIN can display a schema in six different formats. This screen copy shows four of them: text extended (left), text sorted (bottom), graphical compact (top) and graphical standard (right).

7.3.2 DB-MAIN Specification Model and Repository

The repository collects and maintains all the information related to a project. The repository comprises three classes of information:

- a structured collection of schemas and texts used and produced in the wrapper development;
- the specification of the methodology followed to conduct the wrapper development;
- the history (or trace) of the project.

We ignore the second class, which are related to methodological control and which is described in [Roland, 1997]. We will discuss the third class in Section 7.3.7.

A *schema* is a description of the data structures to be processed, while a *text* is any textual material generated or analyzed during the project (e.g. a program or an SQL script). A wrapper development usually comprises several schemas. The schemas of a project are linked through specific relationships. The schema specification is based on a the generic model defined in Chapter 3. Besides the standard concept of the generic model, the repository includes

some meta-objects which can be customized according to specific needs. In addition, annotations can be associated with each object. These annotations can include semi-formal properties, made of the property name and its value, which can be interpreted by Voyager-2 functions (see Section 7.3.3). These features provide dynamic extensibility of the repository. For instance, new concepts such as mapping definition can be represented by specializing the meta-objects, while statistics about entity populations can be represented by semi-formal attributes.

The contents of the repository can be expressed as a pure text file through the ISL language, which provides import-export facilities between DB-MAIN and its environment.

7.3.3 Voyager 2

DB-MAIN offers a complete development language, *Voyager 2* [Englebert, 2001], through which new functions and processors can be developed and seamlessly integrated into the tool. *Voyager 2* offers a powerful language in which specific processors can be developed and integrated into DB-MAIN. Basically, *Voyager 2* is a procedural language which proposes primitives to access and modify the repository through predicative or navigational queries, and to invoke all the basic functions of DB-MAIN. It provides a powerful list manager as well as functions to parse and generate complex text files. A user's tool developed in *Voyager 2* is a program comprising possible recursive procedures and functions. Once compiled, it can be invoked by DB-MAIN just like any basic function.

Figure 7-2 presents a small but powerful *Voyager 2* function which displays some statistics about an ER schema.

```
ne, na, nr;
data_object: d;
integer: typ;
schema: sch;
owner_of_att: own;

/*****
** compute the number of attributes owned
** by a "owner_of_att"
*****/

function integer nbr_att(owner_of_att: o)
attribute: a;
{ return Length(ATTRIBUTE[a]{@OWNER_ATT:[o]});
}
begin
  SetPrintList("", "", "");          /* define the mask to print lists */
  sch:=GetCurrentSchema();          /* what is the current opened schema? */
  if IsVoid(sch) then {              /* oh oh: there is no schema ! */
    print("No Schema !\n");
    halt;                            /* stop here ! */
  }
  /* Initialization */
  ne:=0;
  na:=0;
  nr:=0;
```

```

/* The Body */
for d in DATA_OBJECT[d]{@SCH_DATA:[sch]} do { /* for each data_object in the schema */
  typ:=GetType(d); /* but, what is the type of the data_object */
  switch (typ) {
    case ENTITY_TYPE: /* ... it is an entity_type */
      ne:=ne+1;
      own:=d; /* type-casting of the argument */
      na:=na+nbr_att(own); /* how much attributes in it ? */
    case REL_TYPE: /* ... it is a rel_type */
      nr:=nr+1;
      own:=d; /* type-casting of the argument */
      na:=na+nbr_att(own); /* how much attributes in it ? */
  }
}
print(["\nSTATISTICS:",
      "\n-----",
      "\n#Entity types:\t",ne,
      "\n#Rel-types:\t",nr,
      "\n#Attributes:\t",na,
      "\n"]);
end

```

Figure 7-2: A Voyager 2 program example.

7.3.4 Transformation Toolkit

DB-MAIN proposes a three-level transformation toolset that can be used freely, according to the skill of the user and the complexity of the problem to be solved: namely, elementary transformations, global transformations and model-driven transformations.

Elementary transformations

A schema transformation is applied to the selected construct of a schema:

apply transformation T to current construct C

With these tools, the user keeps full control of the schema transformation. Indeed, similar situations can often be solved by different transformations; e.g., a multivalued attribute can be transformed in a dozen ways. Figure 7-3 illustrates the toolbar for the attribute transformation. The current version of DB-MAIN proposes a toolset of about 30 elementary transformations.

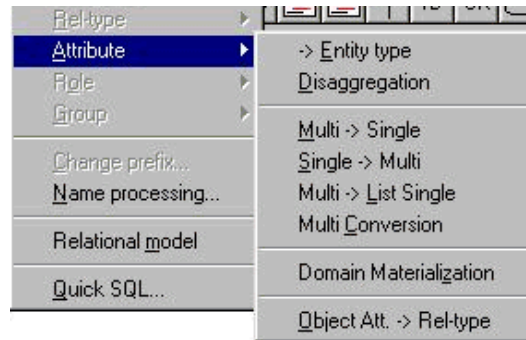


Figure 7-3: Transformation menu of an attribute. For instance, a selected attribute can be transformed into an entity type; a selected compound attribute can be disaggregated.

Global transformations

A selected elementary transformation is applied to all the objects of a schema that satisfy a specified precondition:

apply transformation T to the constructs that satisfy the condition P

DB-MAIN offers some predefined global transformations, such as: *replace all one-to-many relationship types by foreign keys* or *replace all multivalued attributes by entity types*. Moreover, the analyst can define its own toolset through the *Transformation Assistant* described in Section 7.3.6.

Model-driven transformations

All the constructs of a schema that violate a given model M are transformed in such a way that the resulting schema complies to M:

apply the transformation plan which makes the current schema satisfy the model M

Such an operator is defined by the transformation plan described in Chapter 4. DB-MAIN offers a dozen predefined model-based transformations such as relational, CODASYL and COBOL translation, untranslation from these models. The analyst can define its own transformation plans, either through the scripting facilities of the *Transformation Assistant*, or, for more complex problems, through the development of *Voyager 2* functions.

7.3.5 Text Analysis and Processing

This assistant provides a set of sophisticated tools to browse texts such as program source files, to search them for complex text patterns, and to compute abstractions such as dataflow graphs and call graphs. We briefly describe three processors provided by this assistant.

Physical schema extractor. The physical extraction process is carried out by a series of pro-

processors that automatically extract the data structures declared into a source text. These processors identify and parse the declaration part of the source texts, or analyze catalog tables, and create corresponding abstractions in the repository. Extractors have been developed for SQL, COBOL, CODASYL, IMS and RPG data structures. Additional extractors can be developed easily thanks to the Voyager 2 environment.

Interactive pattern-matching engine. The pattern-matching engine searches text files for definite patterns or clichés expressed in PDL, a Pattern Definition Language. This is the main tool to perform usage patterns analysis in programs.

Dataflow graph builder and inspector. This tool is parametrized with the PDL syntactic patterns that define the selected relationships between program variables. The analyst can select a variable A, then examine in context the statements that mention the variables that are connected to A, directly or transitively.

Program slicer. This processor builds the program slice relative to a program point. The program slice can be visualized in context, displayed in selected color, or extracted as an autonomous program on which other tools can be applied, such the pattern-matching engine, the dataflow builder or the program slicer itself.

7.3.6 Assistants

An assistant is a higher-level solver dedicated to coping with a special kind of problems, or performing specific activities efficiently. It gives access to the basic toolboxes of DB-MAIN but in a controlled and intelligent way.

The current version of DB-MAIN includes three general purpose assistants which can support among other, the wrapper development processes, namely, the *Transformation* assistants, the *Schema Analysis* assistant and the *Text Analysis* assistant. These processors offer a collection of built-in functions that can be enriched by user-defined functions developed in *Voyager 2*.

Transformation assistants

The *Transformation Assistant* (Figure 7-4) allows applying one or several transformations to selected constructs. Each operation appears as a problem/solution couple, in which the problem is defined by a pre-condition (e.g., the constructs are the many-to-many relationship types of a schema) and the solution is an action resulting in eliminating the problem (e.g., transform them into entity types). Several dozens problem/solution items are proposed. The analyst can select one of them, and execute it automatically or in a controlled way.

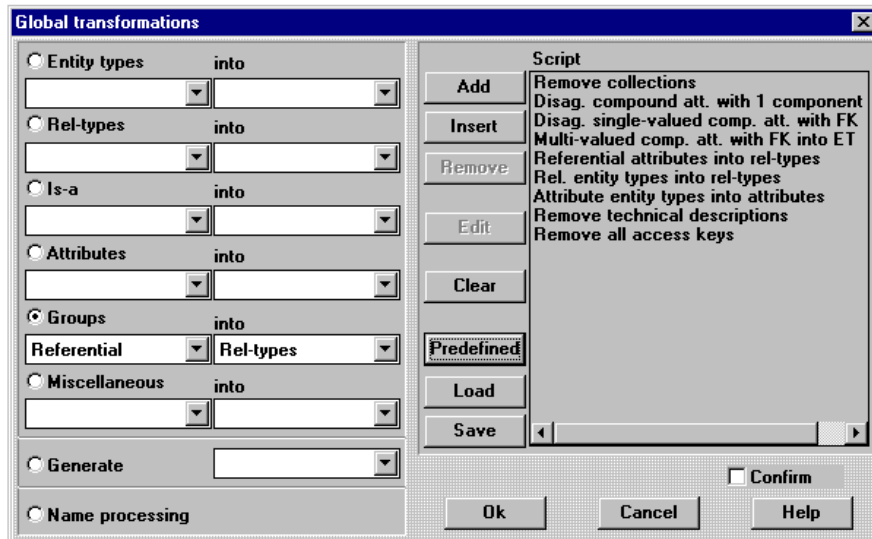


Figure 7-4: The basic global transformation assistant allows the analyst to perform a transformation on all the objects that satisfy a condition (left part). This screen copy shows that the analyst has developed a small script (right part) to conceptualize a COBOL schema. More complex scripts can be developed with the Advanced Global Transformation assistant.

Moreover, the *Advanced global transformations*, a sophisticated version of the *Transformation Assistant*, proves more flexibility and power in script development. A script consists of transformations and control structures. A transformation has the form $A(P)$ where A is an action (transform, remove, mark, etc.) and P is a predicate that selects specific objects in the data schema. The meaning is obvious: apply action A on each object that satisfies predicate P . The control structures include scope restrictions and loops. A library of advanced global transformations can be defined and reused in the definition of new ones.

Example

The *Advanced global transformation Assistant* can be used to build the complex model-driven transformation. Figure 7-5 presents the script of the transformation plan developed in Section 4.6. We recall that the transformation plan has been defined for translating any schema expressed in the wrapper logical model into an equivalent schema expressed in the object-oriented model.

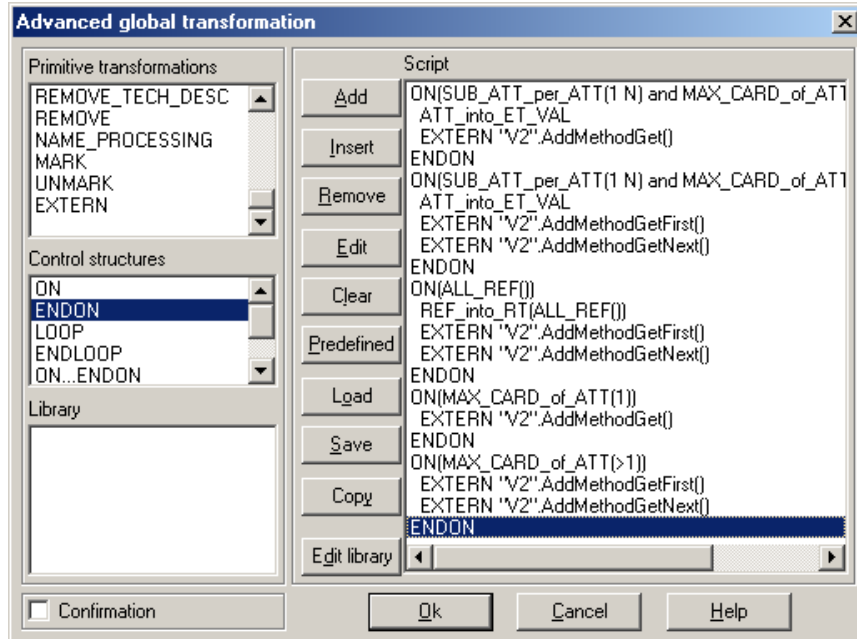


Figure 7-5: The advanced global transformation assistant allows the analyst to write a complex transformation plan like the model translation between the wrapper logical model and the object-oriented model.

Schema analysis assistant

The *Schema Analysis* assistant is dedicated to the structural analysis of schemas. It uses the concept of submodel, defined as a restriction of the generic model (see Chapter 3). This restriction is expressed by a boolean expression of elementary predicates stating which specification patterns are valid, and which ones are forbidden. An elementary predicate can specify situations such as the following: "entity types must have from 1 to 100 attributes", "relationship types have from 2 to 2 roles", "entity type names are less than 18-character long", "names do not include spaces", "no name belongs to a given list of reserved words", "entity types have from 0 to 1 supertype", "the schema is hierarchical", "there are no access keys". A submodel appears as a script which can be saved and loaded. Predefined submodels are available: Normalized ER, Binary ER, NIAM, Functional ER, Bachman, Relational, CODASYL, etc. Customized predicates can be added via *Voyager 2* functions. The *Schema Analysis* assistant offers two functions, namely *Check* and *Search*. *Checking a schema* consists in detecting all the constructs which violate the selected submodel, while the *Search function* detects all the constructs which comply with the selected submodel.

Example

As an illustration, the *Schema Analysis* assistant is able to automatize the detection of all the constructs and constraints that are not available in the wrapper logical model. Figure 7-6 presents the lists of rules that all the constructs must satisfy to comply with the wrapper logical model.

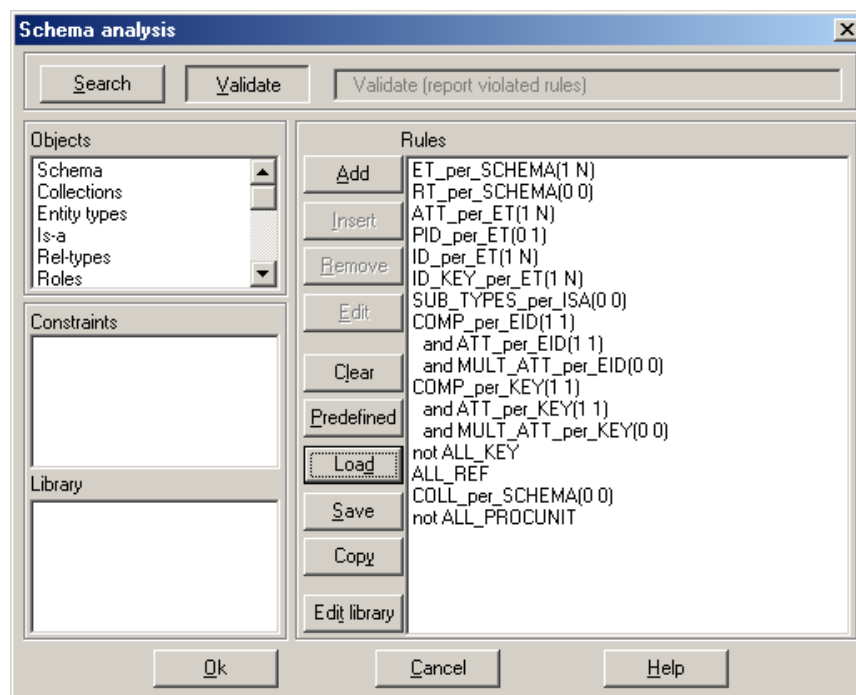


Figure 7-6: The schema assistant allows the analyst to detect the constructs that are not allowed in a specific model.

Foreing key assistant

The *Foreing Key Assistant* proposes some popular heuristics to find foreign keys (as well as inclusion and copy constraints, which generalize the concept of foreign key). The analyst gives a list of groups and chooses one of the two strategies:

- Given a candidate foreign key (in the list of groups), find the possible target record types (a group);
- Given a group (usually an identifier - in the list of groups), find the field (an existing group or an attribute) of the schema that could reference the group.

Depending on the chosen strategy, he gives the criteria to find the matching groups. When the

matching groups are found, he can create the foreign keys.

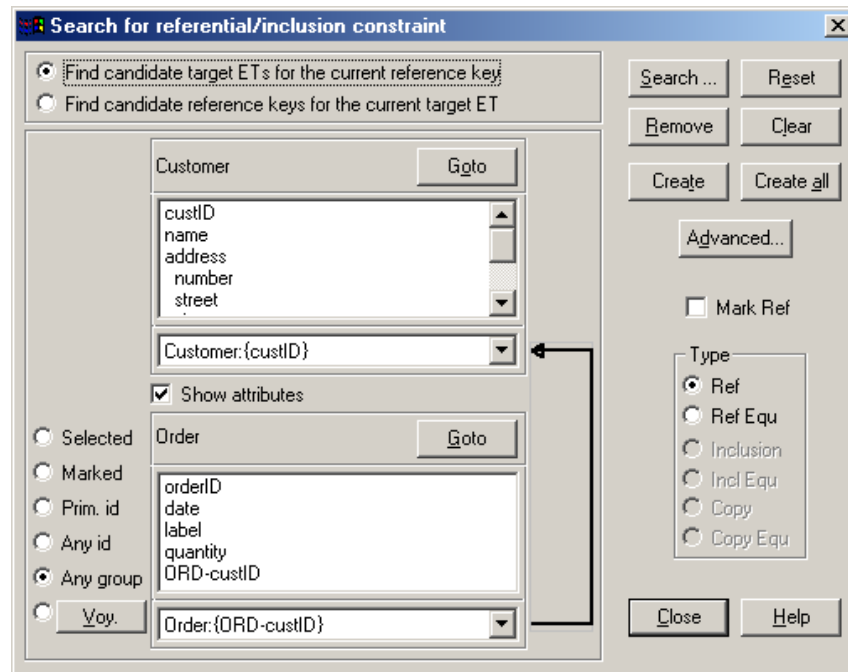


Figure 7-7: The dialog box of the foreign key searching assistant.

7.3.7 History

DB-MAIN automatically generates and maintains a history log (say h) of all the schema transformation that are carried out when the developer derives a schema B from schema A . This history is completely formalized in such a way that it can be replayed, analyzed and transformed. For example, any history h can be inverted into history h' .

If h expresses the structural mapping between the physical and wrapper logical schemas, and if t is the instance mapping of h , then $\{h', t\}$ is the functional specification of the logical wrapper. h' explains how to translate queries while t explains how to form the result instances. Therefore, history h can be used to generate the logical wrapper (see Section 7.4.1).

Example

Figure 7-8 presents a history log of a Att-ET/val transformation: the attribute address of the entity type *Customer* has been transformed into an entity type address. This leads to the creation of the relationship type *has*.

```

*POT "begin-file"
*TRF att_to_et_inst
%BEG
  %NAM "address"
  %OWN 3 "SCHEMA"/"cobol-1"."Customer" 513
  %OID 523
  *POT "##1##gettatt"
  *CRE ENT
  %BEG
    %OID 529
    %NAM "address"
    %POX 126857
    %POY 36776
    %OWN 1 "SCHEMA"/"cobol-1" 476
  %END
  *POT "##1##getrel"
  *CRE REL
  %BEG
    %OID 531
    %NAM "has"
    %SNA "C_a"
    %POX 0
    %POY 0
    %OWN 1 "SCHEMA"/"cobol-1" 476
  %END
  *POT "##0##att_to_et_inst"
%END
*POT "end-file"

```

Figure 7-8: A history log example. The log records information about the transformation of the attribute address into an entity type by value representation.

7.4 InterDB Tools

The InterDB tools have been built within the DB-MAIN environment. They have been developed in *Voyager 2*. The wrapper generation is performed by two specialized tools, namely the *history analyzer* and the *wrapper encoders* (Figure 7-9):

- *History analyzer*. The history analyzer analyses the history *h* in order to enrich the wrapper logical schema of logical/physical semantics correspondences. The end product of this phase is an enriched wrapper logical schema that holds all the information required for the logical wrapper generator.
- *Logical wrapper encoders*. From the enriched wrapper logical schema, the *logical wrapper encoder* produces the procedural code of the specific logical wrapper and a documentation for the programmers whereas the *object wrapper encoder* generates the Java code of the specific object wrapper and the definition of the wrapper object schema

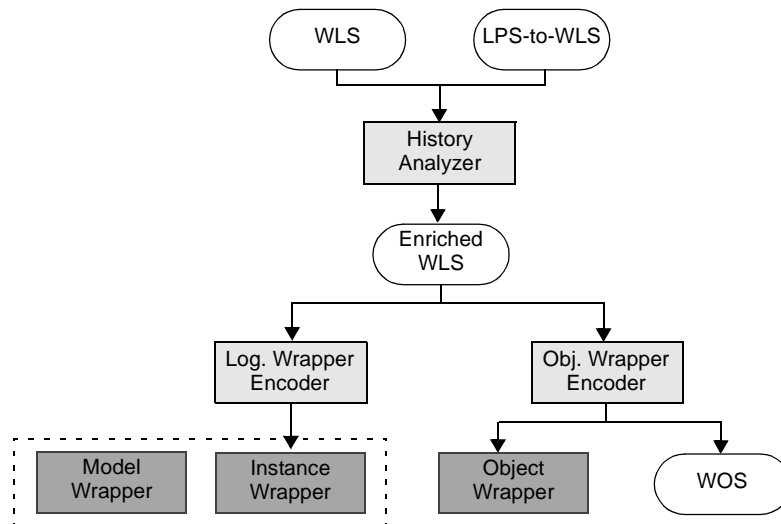


Figure 7-9: InterDB CASE tools: history analyzer and wrapper encoders.

All these tools are built around the DB-MAIN repository (Figure 7-11): the history analyzer extends it by adding meta-objects that represent the correspondence definitions whereas the logical encoders access to it to generate wrapper codes.

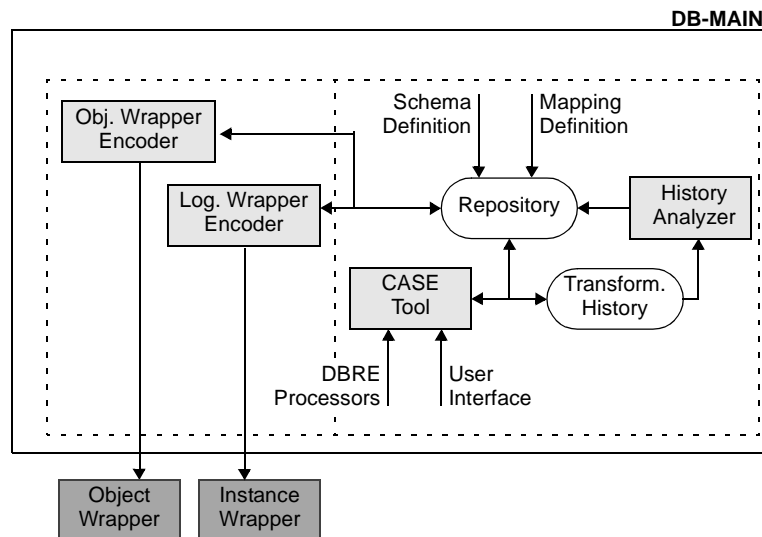


Figure 7-10: History analyzer and wrapper encoders built around the DB-MAIN repository.

7.4.1 History Analyzer

Principles

If h expresses the structural mappings between the physical and wrapper logical schemas and if t is the instance mapping of h , that is, $h=[LPS\text{-}to\text{-}WLS]$ and $t=[lps\text{-}to\text{-}wls]$, then $\{h',t\}$ is the functional specification of the logical wrapper. Therefore, history h can be used to generate the wrapper. However, this form does not provide a good support for reasoning and processing, for which a functional expression is better suited.

The *history analyzer* analyses h in order to transform it into functional specifications from which the wrapper logical schema is enriched with physical/logical semantics correspondences. The end product of this phase is an enriched wrapper logical schema that includes, for each construct, the way it has been mapped onto physical constructs. In this way, this schema holds all the information required for the generators.

Repository extension

The main task of the history analyzer is to extend the DB-MAIN repository with meta-properties so that the DB-MAIN repository can represent the physical/logical semantics correspondence between a wrapper logical schema and the underlying physical schema. The history analyzer proceeds in two main steps:

- It extends the DB-MAIN repository with meta-properties that represent the physical/logical correspondences. A meta-property is defined as a triple $\langle \text{name}, \text{construct}, \text{value domain} \rangle$ that specifies that a construct is associated with a meta-property of value domain value domain. Some of these meta-properties are represented in Figure 7-11.
- It analyzes the history log that holds $\{h',t\}$ and computes, for each schema transformation, the meta-properties of the construct(s) associated to that schema transformation.

Meta Property	Construct	Value
InterDB-file-name	Entity type	to be defined by the user - only for COBOL
InterDB-data-source-name	Schema	to be defined by the user - only for RDB
InterDB-Index	Attribute	true false
InterDB-down-mapping	Attribute Entity type	rename: name at the physical level
InterDB-down-mapping	Compound monov- alued attribute	disaggregate concatenate
InterDB-down-mapping	n-level attribute (n>1)	substring(offset,length)
InterDB-down-mapping	Simple multivalued attribute	M-concatenate M-instantiate
InterDB-down-mapping	Compound multi- valuated attribute	M-instantiate-co-concatenate (<i>liste_attributs</i>)
InterDB-opt-implementation	Optional attribute	user default / system default / null
InterDB-default-str	Schema	to be defined by the user - string as the null value for the string values
InterDB-default-num	Schema	to be defined by the user - number as the null value for the number values
InterDB-FK-implementation	Group	declared / simulated: <i>declared</i> means explicit foreign key; <i>simulated</i> means implicit foreign key (that is emulated by the wrapper)
InterDB-ID-implementation	Group	declared / simulated: <i>declared</i> means explicit identifier; <i>simulated</i> means implicit identifier (that is emulated by the wrapper)

Figure 7-11: Meta-properties defining the mapping properties between the physical and wrapper logical schemas.

Example

To illustrate the history analyzer and the meta-properties it generate, we use the physical and wrapper schemas in Figure 7-12 and consider the history *h*. The mappings between them can easily be deduced in this figure.

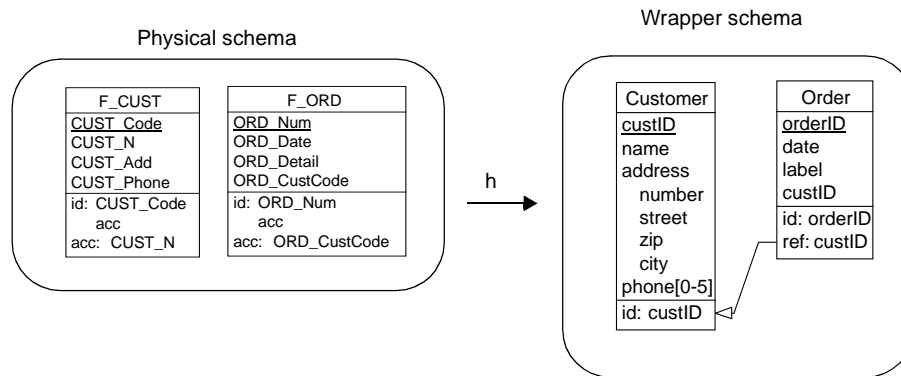


Figure 7-12: Wrapper logical and physical schemas example.

The analysis of the wrapper logical schema and the associated history h gives rise to the meta-property definition of each construct of the wrapper logical schema. In Figure 7-13, we present the main meta-properties and their values of three constructs: the entity type Order; the attribute Phone of Customer and the group CustID of Order.

ORDER (entity type)	
InterDB-down-mapping	rename(F_ORD)

PHONE of CUSTOMER (attribute)	
InterDB-down-mapping	M-Instantiate;5
InterDB-opt-implementation	system-default

CUSTID of ORDER (group)	
InterDB-FK-implementation	simulated

Figure 7-13: Meta-property examples.

7.4.2 Wrapper Encoders

Two kinds of wrapper encoders are available: the logical wrapper encoders and the object wrapper encoder. They both analyze the enriched wrapper logical schema in order to produce their respective codes.

Logical wrapper encoders

The *logical wrapper encoders* produce the procedural code of the logical wrapper and a documentation for the programmers. They also produce the environment data that will allow the communication between the logical wrapper and the logical server (Chapter 5). These encod-

ers are specific to a DMS family (Chapter 6). At the current time, generators for COBOL files and RDB data structures are available.

Object wrapper encoder

The object wrapper encoder generates the interface definition for each object type of the wrapper object schema and their implementation (Chapter 5). The object wrapper is developed in Java on top of the logical wrapper. That is, the object wrapper encoder is independent of the underlying DMS and platform. It is therefore written once for all.

Part III

Mediator Technology

Mediator Architecture

In which the technology of mediators is presented. Their characteristics are outlined and the different levels of service a mediator should manage are presented and discussed. The architecture of the InterDB mediator prototype is finally presented.

8.1 Introduction

The interest of integrated databases has been continuously growing in the last years. Many organizations face the problem of integrating data residing in several distributed databases. Companies that build a data warehouse or an enterprise resource planning system must address the problem. Also, integrating legacy data in the web is the subject of several investigations and projects nowadays.

This problem can be addressed by using *mediators* that offer a virtual and integrated view of legacy and distributed databases. The function of a mediator is to provide integrated information, without the need to integrate the data resources. A mediator hides details about the location and representation of relevant data to applications.

8.2 Mediator Definition

The term *mediator* was introduced by Wiederhold [Wiederhold, 1992]. In general, a *mediator* is a software that mediates between the client applications and (distributed) data sources. It exploits encoded knowledge about some sets or subsets of data to create (integrated) infor-

mation for client applications [Wiederhold, 1992].

As far as database integration is concerned, mediators are considered as software components that obtain information from wrapped databases or other mediators. They provide information to the other mediators above them or to the client applications of the system. The mediator exports a global schema which is an integrated representation of data sources. A mediator can be seen as a view of the data found in one or more data sources. Data are not stored in a mediator. The users query the mediator schema, and the mediator transforms these queries between the mediator schema and the wrapper schemas of the data sources.

Mediator issues

The mediator design is a very complex task which comprises several different issues. The main issues are:

- Dealing with heterogeneity of the sources;
- Specifying the mappings;
- Processing queries expressed on the global schema.

Dealing with heterogeneity of the legacy databases. It refers to the fact that sources adopt different semantics, models and systems for storing data. This issue is partially resolved by wrapping the legacy databases so as to give them uniform and transparent interfaces using a common wrapper model and a common query language (see Chapter 5).

Specifying the mappings. With regard to this issue, two basic approaches have been used to specify the mappings between the wrapper and the global schemas [Li, 2000]. The first approach, called *global-as-view* - GaV (also global-schema centric, or simply global-centric), requires that each construct of the global schema be expressed as a view on the local databases. Several projects like Tsimmis [Chawathe, 1994] and Disco [Tomas, 1996] adopt the GaV approach. In the second approach, called *local-as-view* - LaV (or source-centric), the mappings are defined in an opposite way: each object in a given local database is defined as a view on the global schema. The LaV approach is used in the Information Manifold system [Levy, 1996] and the Infomaster system [Genesereth, 1997]. An example that illustrates and compares LaV and GaV approaches is given in Section 8.4.2.

Processing queries expressed on the global schema. This is concerned with one of the most important problems in the design of a mediator system, namely, the choice of the method for computing the answer to queries posed in terms of the global schema. For this purpose, the system should be able to decompose the query in terms of a suitable set of queries posed to the wrappers. In the reformulation process, the crucial step is to decide how to decompose the query on the global schema into a set of subqueries on the wrappers, based on the meaning of the mappings. The computed subqueries are then shipped to the wrappers, and the results are assembled into the final answer. It is well known that processing queries in the local-as-view approach is a difficult task [Ullman, 1997]. Indeed, in this approach, the only knowledge we have about the data in the global schema is through the views representing the local databases, and such views provide the partial information about the data. Therefore extracting in-

formation for the data integration is similar to query answering with incomplete information. On the other hand, query processing looks much easier in the global-as-view approach, where in general it is assumed that answering a query means unfolding its structures according to their definitions in terms of the sources.

8.3 Architecture

8.3.1 General Framework

The basic idea is that mediators access to legacy data through wrappers. Figure 8-1 shows the general framework that we overview the different aspects of mediators. The framework consists of the following components:

- *Wrapper technology* that attempts to hide the characteristics of the legacy databases from the mediators;
- *Access paradigms* that are used to remotely access the legacy resources;
- *Mediator technology* that offers a unique and integrated view of the legacy databases;
- *New applications* that access to distributed legacy databases through the mediator.

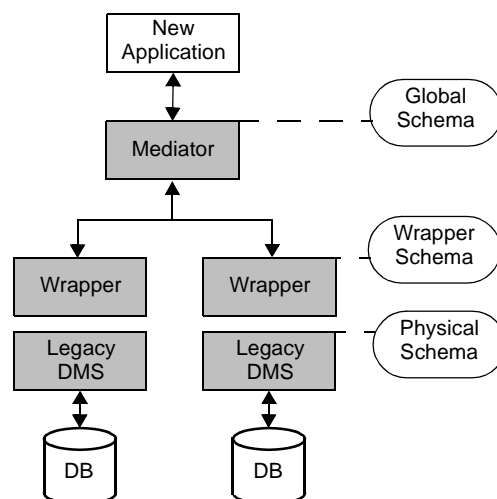


Figure 8-1: Mediator framework.

Wrapped legacy resources

Wrappers are developed on top of legacy databases to give them transparent access interfaces

(see Chapter 5). That is, a wrapper exports and homogenizes the physical schema and the query interface of an underlying database.

Access paradigms

Access infrastructure consists of technologies such as computers, networks and transaction managers. An important part of the platform is *middleware*, an increasingly crucial and, at the same time, bewildering component of the access infrastructure. Middleware services typically include directories, facilities to call remotely located functions and software to access and manipulate remotely located databases. Middleware services are typically provided by specialized software package. However, middleware services may reside in a combination of database management systems, computer operating systems, and transaction management system.

The middleware manages the communications between mediators and wrappers. CORBA [Mowbray, 1995] and RMI [Reese, 1997] are two examples of middleware technologies. CORBA and RMI are two distributed object standards supported by the OMG (Object Management Group). CORBA is an architecture standard for building heterogeneous distributed systems. RMI supports distributed objects written entirely and only in the Java programming language. For the future, it's expected to enable RMI to use the IIOP protocol to communicate with CORBA-compliant remote objects.

By using CORBA, it is possible to encapsulate the wrapper as a set of distributed objects and their associated operations [Dogac, 1995]. These properties provide the means to handle the heterogeneity at platform and location levels, the semantic heterogeneity being solved by the wrapper and the mediator. That is, CORBA allows mediators to communicate with the wrapper without having knowledge of its location.

New applications

New applications are the software components that access to distributed legacy databases through a mediator interface. A new application can be, among others, another mediator or a web application.

8.3.2 Mediator Interface

A mediator is developed on top of legacy databases to give them a transparent and unique access interface. The interface is made up of: (1) an *global schema* of the wrapped databases, expressed in a canonical data model and (2) a *common query language* which uses the semantics defined in the global schema.

Schema interface

The global schema is the result of a *schema integration process* (see Chapter 9). The wrapper schemas expressed in a canonical data model are integrated into one (global) schema ex-

pressed in the same model. As a result, a mediator hides the location and representation of the relevant data to the new applications.

Example

Let us consider a mediator that offers a uniform and global schema resulting of the integration of two wrapper schemas (Figure 8-2). The global schema includes all the semantics of the wrapper schemas. In the wrapper schemas, **ORDER** entity types are similar. Data analysis, i.e., examination of actual instances of the physical data types shows that all the **ORDER.Code** values are in the **ORDER.Number** value set. Therefore, this similarity has been interpreted as a supertype/subtype relation: **ORDER** of source 2 (renamed **BOOK-ORDER**) is made a subtype of **ORDER** of source 1.

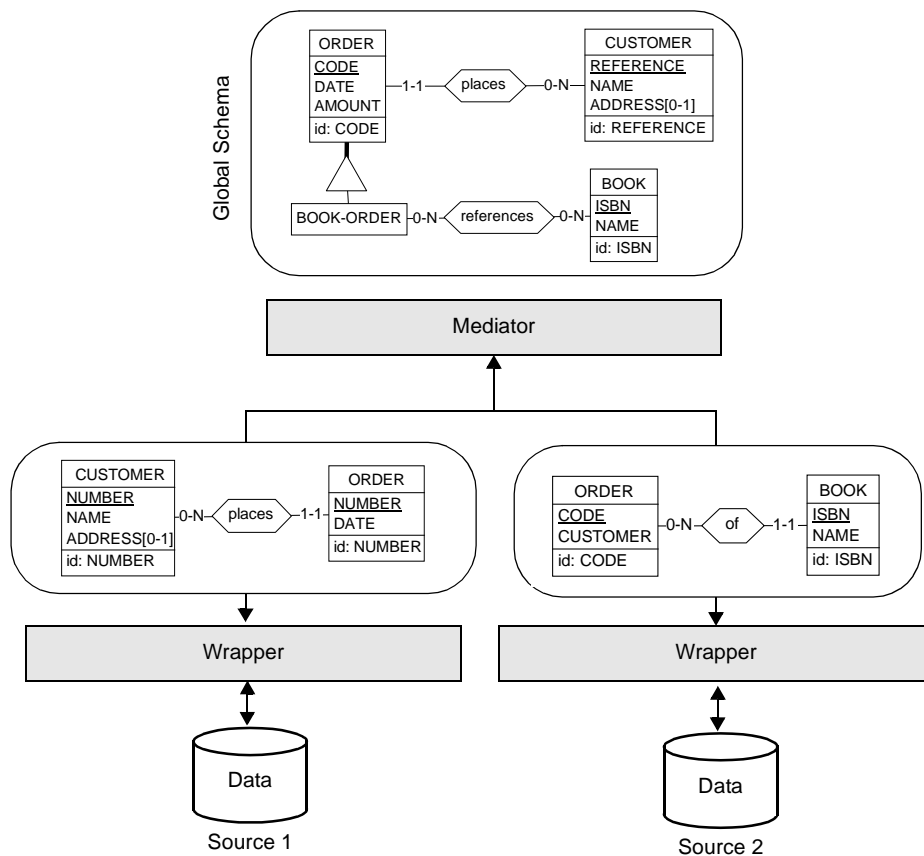


Figure 8-2: Global schema resulting of the integration of two wrapper schemas.

Query interface

The query language of a mediator allows writing queries expressed on global schema constructs, that is, queries addressing the data independently of their distribution across the different sites.

Example

Let us consider the mediator example of Figure 8-3. The mediator query showed in the upper part of Figure 8-3 uses the semantics of the global schema. The query asks for the customers that have placed an order for definite books. The query is decomposed into local queries and local data into global data.

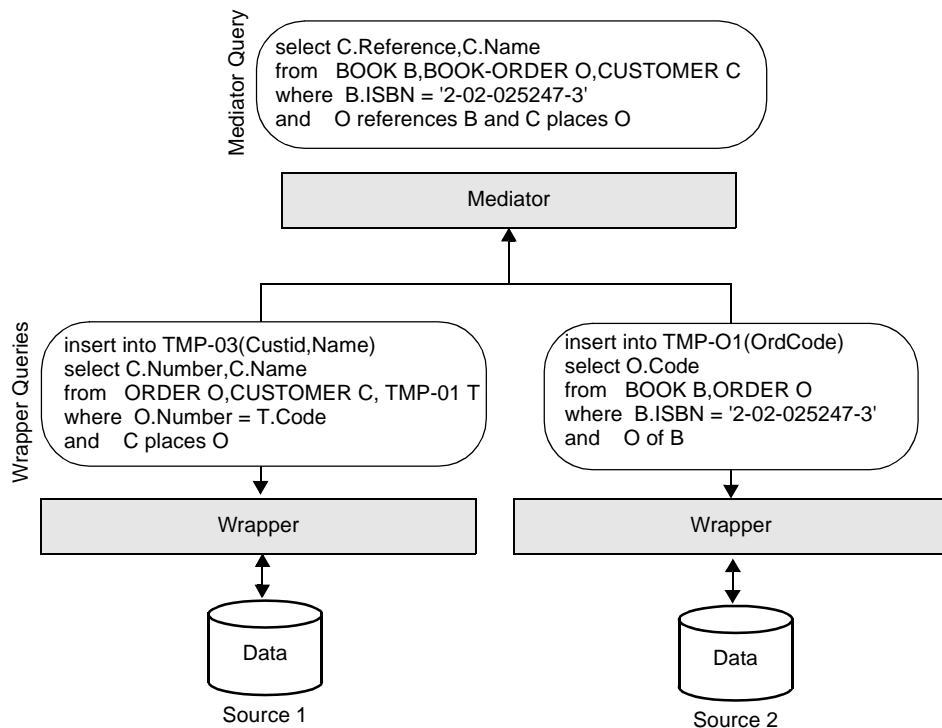


Figure 8-3: Query decomposition example.

Query decomposition

One major mediator service is the *query decomposition* of a mediator query into wrapper queries. A common architecture of query processing is illustrated in Figure 8-4. This architecture

is presented according to the temporal sequence of actions that compose the query processing. Seven actors are identified in this processing [Bouguettaya, 1998]: the query analyzer, the query decomposer, the plan generator, the cost evaluator, the statistics manager, the dispatcher and the execution monitor. We discuss hereinafter the main roles of each of these actors.

A global query is first parsed by the *query analyzer* and then decomposed into subqueries by the *query decomposer*. A subquery usually accesses only a single wrapper and can possibly be composed of one or more of the primitive operations (such as selection, projection, or join on available data at single database site) needed to process a query. The basic principle here is to decompose a query to the finest level in order to explore all possible execution plans.

Given a subquery, the *plan generator* interacts with the *cost evaluator* and the *statistics manager* to generate possible query execution plans and the expected response time and process cost. Subqueries for each query execution plan are formed by grouping adjacent query unit graph together. This grouping process is guided by the cost functions as well as the heuristics which help reduce the search space.

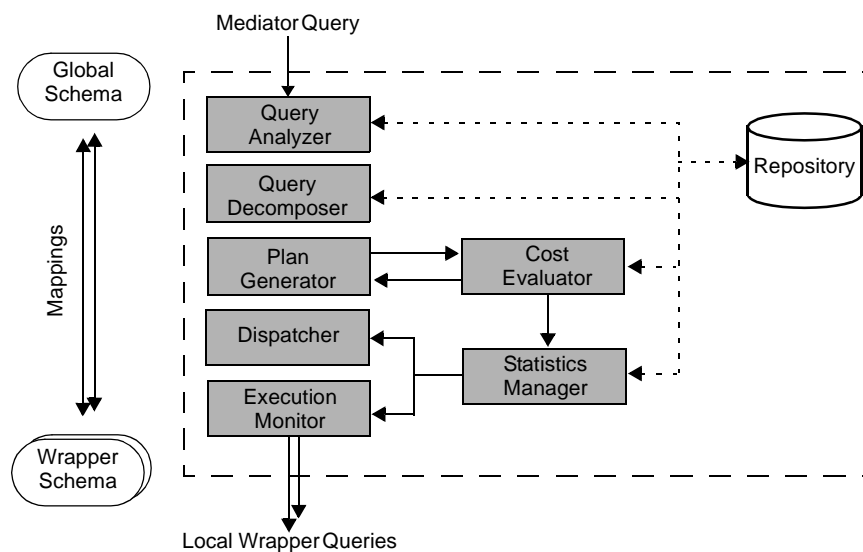


Figure 8-4: Mediator query processor architecture (from [Bouguettaya, 1998]).

The function of the cost evaluator is to provide an estimated cost based on a cost model using statistics managed by the statistics manager. For each global query, there can be a number of query execution plans. Moreover, these plans may have different numbers of subqueries and, hence, different numbers of invocations of wrappers. The parallelism and the size of data to be transferred among the participating wrappers may also differ. Furthermore, the frequency and cost of context mediations will certainly be different as well. All of these differences contribute to different performances of query execution plans which can be generated. The cost

evaluator must take into consideration each of these contributing factors in order to provide an estimated cost. The information related to these contributing factors are kept and managed by the statistics manager. The statistics manager uses the techniques discussed below, such as query sampling and calibration, to obtain such information.

After an optimal plan is chosen, the *dispatcher* dispatches the plan to the relevant site, and the execution monitor coordinates the execution of the chosen plan.

8.4 Mediator Services

Figure 8-5 shows the major services of a mediator, namely, the query analysis, the query processing, the global semantics integrity control and the functionality emulation. To provide these services, a mediator can take advantage of specialized services the wrappers already provide (Chapter 5). We will explain and discuss all these services in the next sections.

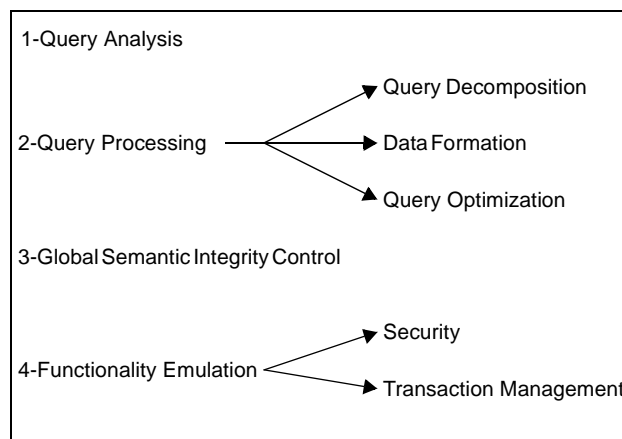


Figure 8-5: Main services of a mediator: (1) query analysis; (2) query processing; (3) global semantic integrity control and (4) functionality emulation.

8.4.1 Query Analysis

The first task of a mediator is the analysis of the input queries. Query analysis enables rejection of queries for which further processing is either impossible or unnecessary [Ozsü, 1991]. The main reasons for rejection are that the query is syntactically or semantically incorrect. When one of these cases is detected, the query is simply returned to the user with an explanation. Otherwise, query processing is continued. A query is incorrect if any of its attribute or entity type names are not defined in the global schema, or if operations are being applied

to attributes of the wrong type.

Example

The following query on a mediator that offers a SQL-like query language and the global schema of the Figure 8-2:

```
select C.Ref-id,C.Name  
from BOOK B, CUSTOMER C  
where B.ISBN = 2020252473  
and O references B
```

is incorrect for two reasons. First; attribute Ref-id is not declared in the schema. Second, the operation =2020252473 is incompatible with the type string of ISBN.

Query analysis can be viewed as three successive steps. First, the query is syntactically analysis. Second, the query is analyzed semantically so that incorrect queries are detected and rejected as early as possible. Techniques to detect incorrect queries use some sort of graph that captures the semantics of the query. Third, the correct query is decomposed into wrapper queries (see Section 8.4.2).

8.4.2 Query Processing

Only in very rare cases, mediator queries against the global schema can not be answered directly by sending them entirely to one wrapper. Instead, the mediator has to find combinations of queries against wrappers that, if combined in a meaningful way, together yield correct results to the original query.

There are four major steps in processing a mediator query: query analysis, query planning, query execution and result integration.

- *Query analysis* is the process of syntactic and semantic analysis of the mediator query (Section 8.4.1).
- *Query planning* is the process of finding a correct plan of executable source queries for a given query against the federated schema. Query planning must be based on predefined correspondences between queries or concepts in different local schemas.
- *Plan execution* is the process of executing a plan. This comprises (1) optimization steps that decide which query operations are performed by which wrapper; (2) the shipping of subqueries and the collection of the results; and (3) applying potentially necessary post-processing such as computing inter-source joins inside the mediator.
- *Result integration* finally tries to homogenize the obtained data by removing redundancy, identifying identical objects and resolving inconsistent data values (i.e. performing fusion integration).

Query planning

Any query planning must be based on some description of the source with respect to the glo-

bal schema, for instance by means of views as defined in SQL. We call the language to express these correspondences *correspondence specification language (CSL)*. Hence, query planning must find correct plans by exploiting the semantic knowledge that is expressed in rules of a certain CSL (e.g., [Catarci, 1993], [Spaccapietra, 1991]). Finding such correspondences is usually the task of a human operator, since they encode the semantic relationships between concepts.

There are two basic classes of CSL. Following the *Global-as-View (GaV)* paradigm, the global construct is defined by having one or more views over the source schemas for each construct. Hence, each correspondence rule has a single global class on one side and defines its semantic equivalence to a source query on the other side (of the rule). The situation is reversed in the *Local-as-View (LaV)* paradigm, where the classes of the source schemas are described by giving equivalent view on the global schema [Hull, 1997]. Again, each rule has a class and a query; but here, it is a local construct and a global query.

The main difference in the perception of the global schema is the following: while GaV sees the global schema as something artificial that must be filled with life by accessing sources, the LaV rather assumes each source as a certain part of overall, global information space.

Global-as-View. Query translation in a GaV approach basically requires the expansion of the structures in a user query into the corresponding source queries [Meng, 1995]. The expansion step "global structure \rightarrow source query" is hardcoded in the definition of the structures (as views). Usually, also the information fusion rules are contained in this definition.

Local-as-View. LaV query planning requires a more complex process, because it is a-priori unclear which parts of a given user query are defined through a view. Every single global view can potentially contribute to a plan for the query. This problem, also known as "answering queries using only views" is shown to be NP-complete already for conjunctive queries and conjunctive view definitions in [Levy, 1995]; it can be solved by enumerating a possibly exponential number of view combinations, and testing query containment for each of these combinations. The problem quickly becomes undecidable, e.g., if negation is allowed.

Example

To illustrate GaV and LaV, we follow an example given by [Lenzerini, 2001]. For simplicity, we consider a relational global schema that is made up of three following relations:

```
movie(Title, Year, Director)
european(Director)
review(Title, Critique)
```

The global schema is the result of the integration of two relational local schemas:

```
r1(Title, Year, Director), in source 1, has recorded the european directors since 1960;
r2(Title, Critique), in source 2, has recorded all the directors since 1990.
```

Assume now that the following mediator query is expressed in the global schema:

$$\{(T, R) \mid \text{movie}(T, 1998, D) \wedge \text{review}(T, R)\}$$

Local-as-View. The local relations are defined in terms of the global schema. That is,

$$r1(T, Y, D) \rightarrow \{(T, Y, D) \mid \text{movie}(T, Y, D) \wedge \text{european}(D) \wedge Y \geq 1960\}$$

$$r2(T, R) \rightarrow \{(T, R) \mid \text{movie}(T, Y, D) \wedge \text{review}(T, R) \wedge Y \geq 1990\}$$

The query $\{(T, R) \mid \text{movie}(T, 1998, D) \wedge \text{review}(T, R)\}$ is processed by means of an inference mechanism that aims at re-expressing the constructs of the global schemas in term of constructs at the sources. In this case:

$$\{(T, R) \mid r2(T, R) \wedge r1(T, 1998, D)\}.$$

Global-as-View. The global schema is defined in terms of the sources:

$$\text{movie}(T, Y, D) \rightarrow \{(T, Y, D) \mid r1(T, Y, D)\}$$

$$\text{european}(D) \rightarrow \{(D) \mid r1(T, Y, D)\}$$

$$\text{review}(T, R) \rightarrow \{(T, R) \mid r2(T, R)\}$$

The query $\{(T, R) \mid \text{movie}(T, 1998, D) \wedge \text{review}(T, R)\}$ is processed by means of unfolding, i.e., by expanding the structures according to their definition in the sources. In this case:

$$\{(T, R) \mid \text{movie}(T, 1998, D) \wedge \text{review}(T, R)\}$$



$$\{(T, R) \mid r1(T, 1998, D) \wedge r2(T, R)\}.$$

Query optimization

In a distributed database system, data is shared across a network of homogeneous databases. As a result of this distribution, the query processor must take into account the locality of information when performing heuristic optimization as well as the cost of transferring data over the network when applying the cost model optimization [Bouguettaya, 1998]. Moreover, it can exploit the potential of parallelism in processing a query.

Query optimization techniques in mediator systems are not significantly different from query processing in distributed database systems [Ozsü, 1991] but more complex ([Sheth, 1989], [Chen, 1998]). How the query optimization paradigm can be implemented in the two environments is very different due the autonomy of the local sites.

Site autonomy. As a result of site autonomy, some local information needed for mediator query optimization may not be available. For example, cost formulas of local DMS, which are essential to global query optimization, are usually not known to the mediator. Even if the cost formulas are available, the mediator still lacks the ability to obtain run-time cost param-

eters, such as data buffer size, that are needed in estimating costs of global subqueries.

Execution autonomy. Execution autonomy means that the mediator cannot influence the way individual DMS process query. The mediator can only interact with it via its wrapper (or its external interface). In other words, the mediator is unable to access internal data structures and functions of the underlying DMS. There is no opportunity for low-level cooperations.

8.4.3 Security Management

A critical feature of mediators concerns the management of confidential data. For that, we need powerful mechanisms for user management, authorization and authentication. Without going into details, we discuss some basic assumptions about security requirements in mediators. Several factors determine these requirements [Schwarz, 1999b]:

- *Autonomy factor.* This means that all local database systems have the right to implement and enforce their own security measures. As a result, the mediator has to respect, to support and to cooperate with the local mechanisms.
- *Heterogeneity factor.* This means, that the local measures rely on different mechanisms and that the local database systems have or more or less trust in a person regarding secrecy and integrity.

As a result, we need *global security measures* which can provide, among others, (1) a *uniform security interface* for all federated users based on a locally accepted federated security policy; and (2) approaches to *resolve security conflicts* that arise from the syntactic and the semantic heterogeneity among the local systems.

The identification and authentication of users in mediators is a more complex process than in traditional distributed systems [Schwarz, 1999b]. We now discuss the main reasons and the resulting tasks.

Heterogeneity. The local authentication components can base on the variety of security concepts. As a result, users have to pass through all these different procedures to gain access. Besides, the identity of a user can vary from system to system. The task is to overcome this heterogeneity without a violation of security. That means, each user is authenticated once but correct to all relevant participating systems per session in a federated environment.

Autonomy. The local authentication decision is dependent on the delivery of the correct identifier about a user access wish. The maintenance of this kind of autonomy can be necessary to secure the trust of the local database systems.

Population control. A user can operate in a mediator with different identities and identifiers. Otherwise, a user should be handled as a single subject independent of the identify he logs in. The authentication of users without local identities can be a task in environments with closed local populations.

Example

There exist many works in the area of authorization for mediators and, more generally,

for federated database systems. We refer to [De Capitani, 1997] for a theoretical introduction and we refer to [Schwarz, 1999] for a practical approach.

8.4.4 Global Semantic Integrity Management

The global schema of a mediator reflects all local integrity constraints. That is, local integrity constraints must be integrated (see Chapter 9). For instance, when two local entity types are merged into one common global entity type, the corresponding integrity constraints must be merged in an appropriate way. Afterwards, new global integrity constraints can be defined on the global schema [Türker, 1999]. They can be used to semantically enrich the global schema. In general, the global integrity constraints a mediator can manage are the following ([Türker, 1999], [Castellanos, 1994]): global uniqueness constraints, global referential constraints and global value dependencies. We hereafter present these global integrity constraints and illustrate them by examples given by [Türker, 1999].

Global uniqueness constraints

A global uniqueness constraint is defined on an entity type or a relationship type of the global schema. Since such a global construct can be defined from several local constructs, a global uniqueness constraint can concern constructs of different wrapper schemas. Often, global uniqueness constraints have to be introduced due to semantic heterogeneity [Garcia, 1996].

Example

Assume that there are two departments managing the information about their projects in own local databases. Each of these databases contains the entity type *Project*. Suppose that these entity types are integrated into a common entity type *Project* which represents all projects of the both departments. In order to guarantee that the projects are uniquely identified in the context of the global schema, the project identifier of each project has to be unique. To ensure this, the mediator has to manage a global uniqueness constraint.

Global referential constraints

Global referential constraints are used to describe existence dependencies between entity types of different wrapper schemas. A global referential constraint expresses the following statement: an entity type ET_1 of a wrapper schema of a local database DB_1 cannot exist unless another entity type ET_2 exists in a wrapper schema of another local database DB_2 . In this case, the entity type ET_1 is called *existence dependent* on the entity type ET_2 .

Example

For instance, in one local database, all the sales information about the products of the company is stored and managed whereas, in another local database, the technical information about the products is available. In this case, we can define a global referential

constraint stating that for each product in the sales database they must exist a product in the technical database. In this way, we can ensure that the sales database only contains products that are produced by the company.

Global value dependencies

A global value dependency concerns the value correspondence between two entity types of different wrapper schemas. An entity type ET_1 is called *value dependent* on another entity type ET_2 if any changes in the attribute values of the entity type ET_1 cause changes to the corresponding attribute values of the entity type ET_2 .

Examples

The address of an employee must be equal in both the local databases. If the address is changed on one database, this change has to be propagated to the other database.

Another example of global value dependencies is "*the number of running projects must not exceed the number of employees*".

Global semantic control by the mediator

Few solutions exist for handling global integrity constraints in (heterogeneous) distributed systems [Özsu, 1999]. The main reason is that semantic data control can be prohibitive. The two main issues for efficiently performing global semantic control are the definition and storage of the rules (site selection) and the design of enforcement algorithms which minimize communication costs. The problem is difficult since increased functionality tends to increase site communications. The reader interested about implementations of global semantic control is invited to read [Özsu, 1999].

8.4.5 Transaction Management

Transaction management is probably the major open question in mediator systems. The challenge is to permit concurrent global updates to the underlying databases through their wrappers without violating the database autonomy. Most current prototypes either do not permit updates to the local databases or execute updates off-line and in batch mode. Recently, much attention has been focused on providing support for updates that span multiple autonomous database systems.

Distributed transaction architecture

Let us first focus on the architectural aspects of mediation transaction processing (Figure 8-6). The federated database architecture involves a number of wrappers, each with its own transaction manager (called *wrapper transaction managers* or WTM) and a mediator on top. A wrapper can use the transaction manager features of the underlying DMS (called *legacy transaction managers* or LTM) if such exists. The transaction manager of the mediator is

called the *global transaction manager* (GTM) since it manages the execution of global transactions.

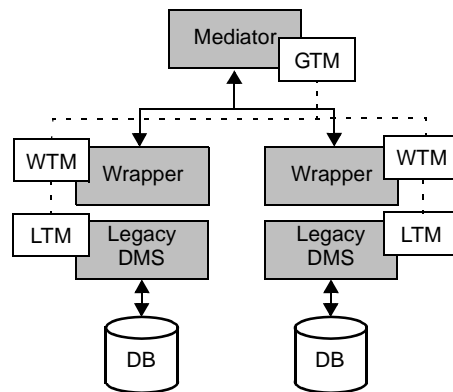


Figure 8-6: Transaction managers in a wrapper/mediator architecture.

Since the wrapper hides the legacy DMS interface (and therefore LTM), we'll hereafter only consider two types of transaction managers: WTM, which are submitted to each wrapper, and GTM, which is submitted to the mediator. Local transactions execute on a single database through a wrapper, whereas global transactions access multiple databases through their wrappers. A global transaction is divided into a set of global subtransactions, each of which executes on one wrapper.

In such an execution environment, it is necessary to discuss the responsibilities of the local and global transaction managers. A series of conditions have been defined that specify when global transactions can safely update a federated system [Gligor, 1986]. These conditions are helpful in determining the minimal functionality required of the various transaction managers.

The first condition for providing global concurrency control is to have wrappers guarantee *local synchronization atomicity*. This means that the wrapper transaction managers are simply responsible for the correct execution of the transactions on their respective databases. If serializability is the correctness criterion used, each wrapper transaction manager is responsible for maintaining that its schedule is serializable and recoverable. These schedules are made up of global subtransactions as well as local ones (local applications). The fundamental point to watch out for here is that the wrapper accepts a transaction and executes it until its termination.

The second condition requires each WTM maintains the relative execution order of the subtransactions determined by the GTM. The global transaction manager then is responsible for coordinating the submission of the global subtransactions to the WTM and coordinating their execution.

Mediation transaction issues

Designing a concurrency control strategy for a heterogeneous database environment is more difficult than designing one for its homogeneous counterpart [Ozsü, 1991], because we must deal with the data distribution but also with heterogeneity and autonomy of the underlying databases. Such a system must deal with problems caused by the autonomy of the local databases:

- *Design autonomy*: local transaction managers are designed in such a way that they are totally unaware of other local database systems and of the integration process.
- *Communication autonomy*: the global transaction manager needs information about local executions in order to maintain global database consistency. However, the GTM has no direct access to this information and cannot force the local concurrency controllers to supply it.
- *Execution autonomy*: local transaction managers make decisions about transaction commitments based entirely on their own considerations. They don't know or care whether commitment of a particular transaction will introduce global database inconsistency. In addition, the GTM has no control over the LTM at all. For example, a GTM cannot force a LTM to restart a local transaction, even if the commitment of this local transaction will introduce global database inconsistency.

DMS Heterogeneity adds further difficulty since it becomes difficult to make uniform assumptions about the functionality and the efficiency provided by the underlying DMS. However, wrapper can partially hide the DMS heterogeneity by providing a uniform transaction manager (WTM) (Chapter 5).

Global transaction protocols

In the literature, a variety of transaction protocols have been proposed, most of which are based on *2PC protocol* (two-phase protocol). The 2PC protocol extends the affects of local atomic actions to distributed transactions by insisting that all sites involved in the execution of a distributed transaction agree to commit the transaction before its effects are made permanent [Samaras, 1995]. A brief description of the 2PC protocol that does not consider failures is as follows [Reddy, 1998]. Initially, the GTM writes a BEGIN-COMMIT record in the log, sends a PREPARE message to all participating sites, and enters the wait state. When a wrapper receives PREPARE message, it checks if it can commit the transaction. If so, the participant writes a ready record in the log, sends a VOTE-COMMIT to the GTM, and enters the ready state. Otherwise, the participant writes an abort and sends a VOTE-ABORT message to the GTM. If the decision of the wrapper is to abort, it can forget about the transaction. The coordinator aborts the transaction globally, even it receives VOTE-ABORT message from one wrapper. Then it writes an abort record, sends a GLOBAL-ABORT message to all participant sites, and enters the abort state; Otherwise, it writes a commit record, sends a GLOBAL-COMMIT message to all sites, and enters the commit state. The participants either commit or abort the transaction according to the GTM instructions and sends back ACK (acknowledgment)

message at which point the GTM terminates the transaction by writing an END-TRANSACTION record in the log.

8.5 InterDB Prototype

An InterDB mediator server appears as a virtual OO interface layer defined by the federated object-oriented schema (FOS). OO views provide transparent read-only access to data sources from wrapper and other mediator servers.

For flexibility reason, the InterDB mediator is based on the DB-MAIN repository that describes the federated object-oriented schema, the local object-oriented schema of each wrapper, its location, and the relationships between local and global schemas. Information concerning data replication, semantic conflicts and data heterogeneity allows the InterDB mediator to interpret and distribute the global objects, and to collect and integrate the results sent back by the wrappers. The extension of the DB-MAIN repository is presented in Chapter 10.

8.5.1 Mediator Architecture

The InterDB mediator architecture as illustrated in Figure 8-7 consists of the following main components:

- the object mediator that links the global objects to the local objects provided by the object wrappers (see Chapter 5);
- the DB-MAIN repository that is accessed by the object mediator through the JiDBM interface (see Chapter 10);
- RMI that supports the communications between object wrappers and mediators and between mediators and client applications.

That is, the object mediator offers remote global objects (FOS) that can be accessed by any Java client applications. Local objects (WOS) that participates in the definition of the global objects must be homogenized against FOS.

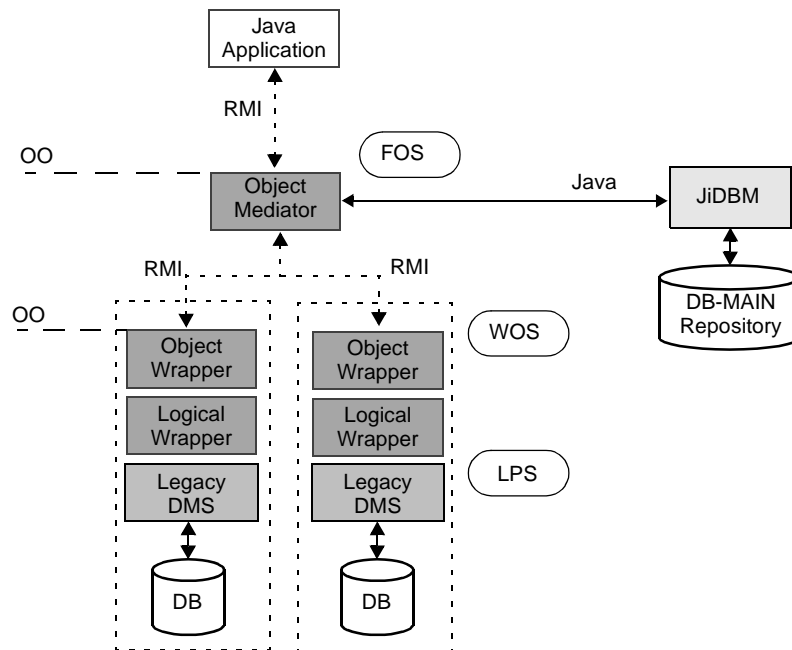


Figure 8-7: InterDB Java Mediator based on the DB-MAIN Repository.

8.5.2 Object Mediator and DB-MAIN Repository

The InterDB object mediator is based on a simple object-oriented model (Chapter 3). Object types describe the definition of objects by their attributes, properties and meta-properties (Chapter 10). We recall that the definition of all the object types are recorded in the DB-MAIN repository. More details about the extension of the DB-MAIN repository can be found in Chapter 10.

The object meta-properties specify the source objects by using operators like selection, union and join. They also specify mappings between local and global attributes if required. An attribute mapping can be described in the following variants:

- *without an explicit mapping*: a local attribute corresponding to an attribute defined by the global object in the FOS in terms of identifier and type becomes an attribute of the global object.
- *with a renaming mapping*: $\text{localName} \leftarrow \text{rename}(\text{globalName})$ means renaming the local attribute to globalName .

- *with a functional mapping:* $\text{localName} \leftarrow \text{function}(\text{globalName})$ defines that the global attribute value is calculated by using the *user-defined conversion function* on the local attribute value.

The union and join operators must be specified with the comparison attributes for union operators or the comparison expression for join operators. Both the union and join operators can be applied with a *user-defined function* for the conflict resolution.

User-defined functions are manually implemented in Java and registered in the mediator system. They are used as conversion functions for attributes or as reconciliation functions for resolving conflicts.

Example

Figure 8-8 depicts a global object type (FOS) containing product information. This object type is the result of the integration of two local object types (WOS₁ and WOS₂). More precisely, the global object type is defined as the union of the two local object types using the product number (Npro) as the matching attribute.

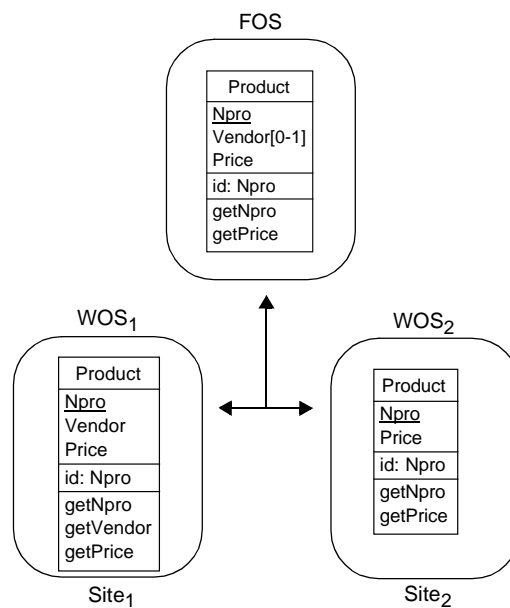


Figure 8-8: Local and global object example.

The two local object types hold some conflicts: (1) in WOS₁, the price data is recorded in Euro while in WOS₂, the same data is recorded in Belgian Franc; (2) in WOS₁, the object type includes the attribute vendor that is not took up in WOS₂; (3) For a same product number (Npro), the prices can be different according to the sources. That is, we assume

that the prices in Euro (from WOS_1) are the most up-to-date prices. Therefore, the price in Euro should be used if there are two different prices for a same product.

The integration mappings are translated as meta-properties of the global object type and its attributes. Some examples of meta-properties for the global object type `Product` and one of its attribute (`Price`) are given in Figure 8-9.

FOS.Product (object type)	
Operator	Union
Local objects	WOS_1 .Product WOS_2 .Product
Matching attributes	WOS_1 .Product.Npro WOS_2 .Product.Npro
Reconciliation function	<code>resolveConflict(WOS_1.Product, WOS_2.Product)</code>

FOS.Product.Price (attribute)	
Local attributes	WOS_1 .Product.Price WOS_2 .Product.Price
Conversion function	<code>BelgianFranc2Euro(WOS_2.Product.Price)</code>

Figure 8-9: Meta-property examples.

In Figure 8-9, we point out that `Product` includes a reconciliation function `resolveConflict` (Figure 8-10) whereas `Price` specifies a functional mapping `BelgianFranc2Euro` (Figure 8-11). We recall that all these functions are manually written in Java.

```
GlobalObject resolveConflict(LocalObject i1, LocalObject i2) {
    GlobalObject res = new GlobalObject();
    // resolve price and vendor attribute:
    if (i1 != null) {res.setString("Vendor", i1.getString("Vendor"));
                    res.setFloat("Price", i1.getFloat("Price"));
                    res.setString("Npro", i1.getString("Npro")); } // copy npro

    else          {res.setString("Vendor", null);
                    res.setFloat("Price", i2.getFloat("Price"));
                    res.setString("Npro", i2.getString("Npro")); } // copy npro

    return res;
}
```

Figure 8-10: Reconciliation function example.

```
Integer BelgianFranc2Euro(Integer price) {
    Integer priceEuro = new Integer();
    priceEuro = price / 40,3399;
    return priceEuro;
}
```

Figure 8-11: Attribute functional mapping example.

8.5.3 Algorithm Principle of the Object Mediator

This section presents the example-driven algorithm principle of the object mediator. The main idea is to present the main mechanisms for data merging and conflict resolution.

Attribute and matching operators

For understanding the mediator algorithm principle, we introduce the concept of *Global Identifier (GID)*. It is the identifier of a global object. The GID is used by the mediator for object matching, to identify instances for local sources that describe the same object so that they can be combined to form instances in the global schema.

The mediator uses the attribute and matching operators (union, join) to build the instances of the global object using the corresponding local objects based on GID values.

Example

To illustrate the mediator algorithm principle, we use the example of Figure 8-8. The GID of FOS is Npro.

Assume that the WOS₁ and WOS₂ represent instance sets as shown in Figure 8-12. By applying the attribute operators (BelgianFranc2Euro defined on Price of WOS₂) and the union operator between WOS₁ and WOS₂, the mediator derives instance of FOS as shown in Figure 8-13.

WOS ₁ .Product		
Npro	Vendor	Price (euro)
1	Michaux	123
2	Gide	10

WOS ₂ .Product	
Npro	Price (BEF)
1	4420
2	400
3	4000

Figure 8-12: Instance sets of WOS₁ and WOS₂

FOS.Product		
Npro	Vendor	Price (euro)
1	Michaux	123
1	null	109.58
2	Gide	10
2	null	9.92
3	null	99.16

Figure 8-13: Derived instances of FOS.

Instance conflict resolution

In Figure 8-12, WOS_1 indicates that the product 1 is sold at 123 euro while WOS_2 indicates that the same product is sold at 109.58 euro. This conflict is reflected in Figure 8-13 as a violation of the GID since they are more than one instance with $Npro=1$. These instances form the *Alternative Instance Set* for $Npro=1$, denoted as $AIS(FOS.Product, 1)$. An alternative instance set containing more than one distinct value indicates an instance level conflict.

Example

For example, we have the following AIS in Figure 8-13:

$AIS(FOS.Product, 1) = \{(1, Michaux, 123), (1, null, 109.58)\}$

$AIS(FOS.Product, 2) = \{(2, Gide, 10), (2, null, 9.92)\}$

$AIS(FOS.Product, 3) = \{(3, null, 99.16)\}$

As a result, there are conflicts with $Npro=1$ and $Npro=2$.

At this point of the algorithm, the mediator has not yet removed any instance conflicts. The conflict resolution is performed by applying the reconciliation function `resolveConflict` for each $|AIS| > 1$. In other words, the reconciliation function resolves the conflict values for all the non-GID attributes of FOS over which there may exist conflicts.

Example

The reconciliation function `resolveConflict` is applied for $AIS(FOS.Product, 1)$ and $AIS(FOS.Product, 2)$. That is, `resolveConflict` (Figure 8-10) resolves conflicts on attribute Vendor and Price of FOS.Product. The result is given in Figure 8-14.

FOS.Product		
Npro	Vendor	Price (euro)
1	Michaux	109,58
2	Gide	9,92
3	null	99,16

Figure 8-14: Conflict-free instances of FOS.

Mediator Development

In which we present an overview of database integration. The issues are raised and the approaches that have been proposed to tackle the problem are discussed. The InterDB approach is then presented and its main characteristics are outlined.

9.1 Introduction

Mediator provides an homogeneous interface to distributed and heterogeneous databases. This homogeneous interface consists of a *global (or federated) schema* which is the result of the integration of the schemas of the corresponding local databases. Simply stated, *database integration* is the process which takes as input a set of databases, and produces as output a single unified description (the global schema) of the input schemas and the associated mapping information supporting integrated access to existing data through the integrated schema [Parent, 1998]. Schema integration is a complex and time-consuming problem ([Heimbigner, 1985], [Parent, 1998], [Elmagarmid, 1999]), primarily because the same fact may be contained in several databases yet be represented using different conceptual structures. A main problem in schema integration therefore concerns the detection and resolution of semantic heterogeneity.

The rest of this chapter is organized as follows. Section 9.2 presents a framework for schema integration. Section 9.3 discusses the main issues of schema integration and their possible solutions. In Section 9.4, we describe the InterDB approach in schema integration in detail.

9.2 Framework for Schema Integration

This chapter provides a survey of most significant trends in schema integration. To provide a framework for schema integration, we first outline the main steps of a typical methodology and then present the main issues of schema integration. A typical schema integration methodology can be divided into four phases. The steps shown in Figure 9-1, are as follows:

- *Pre-integration* where local schemas are transformed to make them homogeneous (both syntactically and semantically);
- *Correspondence identification* devoted to the identification and categorization of interschema relationships;
- *Schema integration* which solves interschema conflicts;
- *Mapping definition* involves storing information about the mappings between constructs in the transformed (global/integrated) schema (FCS) and constructs in the local schemas (LCS).

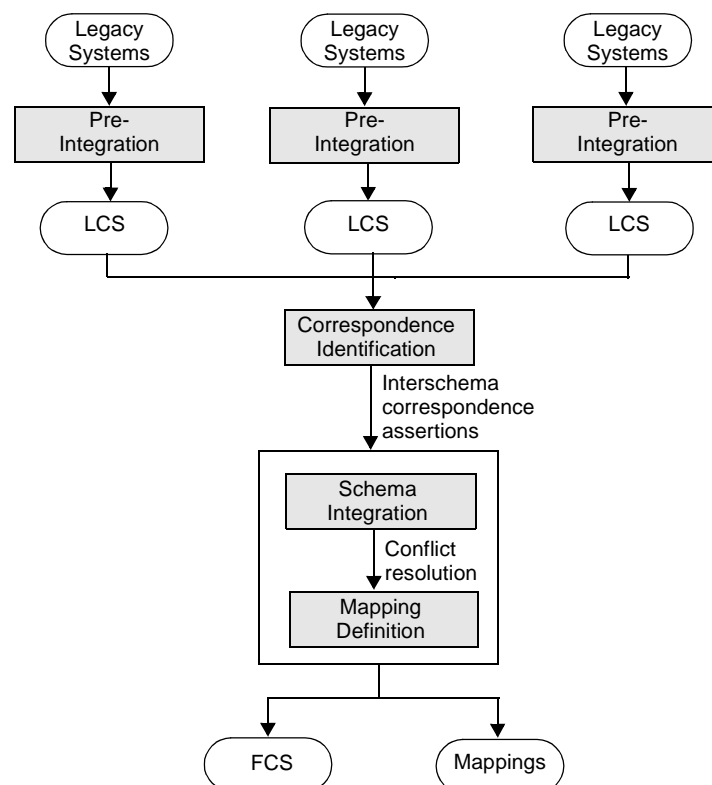


Figure 9-1: Framework for schema integration: the main steps.

In this section, we take up and develop the example that has been introduced in Chapter 6. We recall that this example comprises two independent heterogeneous databases both describing aspects of a bookshop. The first one is made up of two COBOL files and the second one includes two relation tables (Figure 9-3). Through this example, we have already illustrated some of the problems of the pre-integration phase (i.e. semantic enrichment and syntactic rewriting). This example is the starting point for the illustration of the integration processes.

9.2.1 Integration Strategies

In this phase, policies and rules of integration are set. Integration methodologies can be classified as binary or *nary* mechanisms [Batini, 1986] (Figure 9-2):

- *Binary integration methodologies* involve the manipulation of two schemas at a time. These can occur in a stepwise fashion where intermediate schemas are created for integration with subsequent schemas or in a purely binary fashion where each schema is integrated with one other, creating an intermediate schema for integration with other intermediate schemas.
- *Nary integration methodologies* integrate more than two schemas at each iteration. One-pass integration occurs when all schemas are integrated at once, producing the global schema after one iteration.

The binary strategy is simple and efficient, but may lose global information during each intermediate step because not all the information is available at the same time. Although the one-pass integration is complex, it benefits of the availability of complete information about all the databases at integration time. Moreover, there is not implied priority for the integration order of schemas, and the trade-offs, such as the best representation for data items or the most understandable structure, can be made between all schemas rather than between a few.

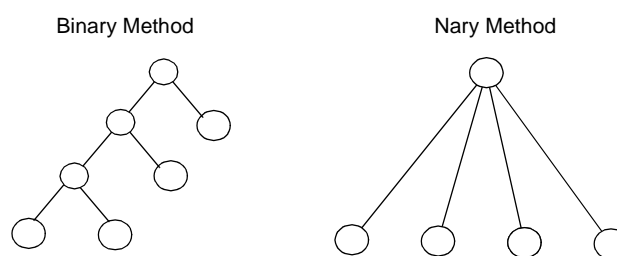


Figure 9-2: Binary and nary method examples.

9.2.2 Pre-integration

In this phase, schemas that correspond to the individual databases being integrated are trans-

lated into schemas using a canonical data model. This phase partially corresponds to the logical wrapper development presented in Chapter 6. We recall that the pre-integration process includes two main tasks, namely, the syntactic rewriting and the semantic enrichment.

Syntactic rewriting. Local schemas are translated into a canonical data model. This allows for resolving syntactic heterogeneity this is the result of different data models.

Semantic enrichment. This is the process that aims at augmenting the knowledge about the semantics of data. Extracting a semantically rich description from a data source is the main goal of the data reverse engineering process (DBRE). Reverse engineering relies on the analysis of whatever information is available: schema specifications, index definitions, data, queries in existing programs. Combining these analysis makes it possible to recover hidden structures and constraints [Hainaut, 1996].

Example

Figure 9-3 shows the extracted schemas of the relational database and of the COBOL files according to their data model.

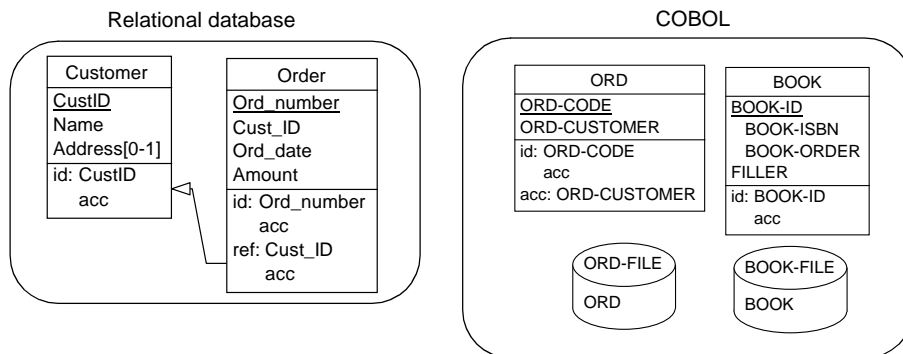


Figure 9-3: The local physical schemas of the relational database (left) and of the COBOL files (right).

The physical schemas of Figure 9-3 have been translated into a canonical data model (the generic model) and they have been enriched with implicit constraints and constructs (Figure 9-4).

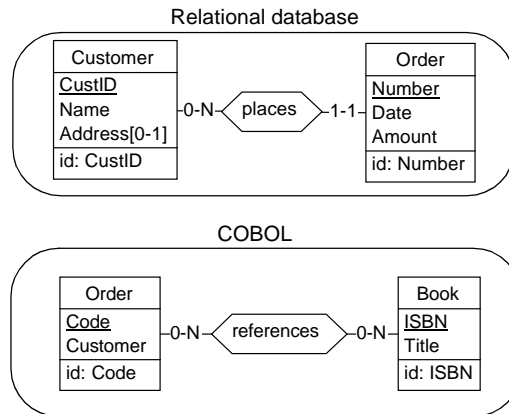


Figure 9-4: The local conceptual schemas (LCS) of the relational database (up) and of the COBOL files (down).

9.2.3 Correspondence Identification

The objective of this phase is to identify objects in the underlying schemas that may be related and to categorize the relationships among them. This is done by examining the semantics of the structures in the different databases and identifying relationships based on their semantics. The semantics of an object can be ascertained by analyzing schematic properties of entity types, attributes, and relationships in the schema as well as by interacting with designers and exploiting their knowledge and understanding of the application domain. For example, integrity constraints, cardinality, and domains are properties of attributes that convey their partial semantics. The ultimate objective of this step is the generation of a reliable set of relationships among database constructs. It is important that these relationships be accurate because they are used as input to the schema integration phase.

Example

For example, in Figure 9-4, we identify the two entity types **Order** in the relation database and **Order** in the COBOL files are related to each other. Moreover, we identify the attributes **Code** and **Number** in the two entity types are related.

9.2.4 Schema Integration

In this phase, the interschema relationships generated previously are used to generate an integrated representation of the underlying schemas. Generating such a representation involves resolving various forms of heterogeneity that may exist between related constructs. [Sheth,

1993b] classifies these heterogeneities into five major categories: domain definition, entity definition, data value, abstraction level and schematic incompatibilities. The integrated schema generation process resolves these different forms of heterogeneity and generates an integrated schema that hides the heterogeneity from the user.

Taxonomies of conflicts abound in the literature, from very detailed ones [Sheth, 1992] to simpler ones [Spaccapietra, 1991]. Some examples of well-known conflict categories are:

- *Heterogeneity conflicts*: different data models support the input schemas;
- *Generalization/specialization conflicts*: related databases represent different view-points on the same set of objects, resulting in different generalization/specialization hierarchies, with objects distributed according to different classification abstractions;
- *Description conflicts*: the related types have different sets of properties and/or their corresponding properties are described on different ways;
- *Structural conflicts*: the constructs used for describing the related types are different;
- *Data conflicts*: corresponding instances have different values for corresponding properties.

Example

In our example, in the integrated schema, an equivalent relationship between the entity types Order is generated to reflect the nature of the relationship among these entity types. Note that the attributes Number and Code have been integrated into a single attribute (Code) in the super entity type. This type of integration assumes that these attributes have been identified as being equivalent to each other in the interschema relationship generation step.

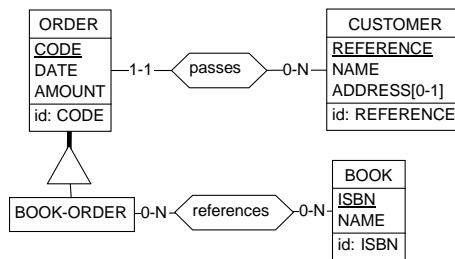


Figure 9-5: Global conceptual schema.

9.2.5 Mapping Definition

This step accompanies the integrated schema generation step and involves storing information about the mappings between structures in the transformed (global/integrated) schemas and structures in the (local) wrapper schemas. Such mappings are important for query transformation. It should be noted that these steps may need to be performed iteratively to resolve

the heterogeneity and arrive at an integrated representation of the underlying schemas.

Example

For example, we note that the attribute `Code` in the integrated schema maps back to `Code` in the conceptual schema of the Cobol files and `Number` in the conceptual schema of the relational database.

9.3 Schema Integration Issues

The purpose of this section is to provide a clear picture of what are the main issues and the current solutions in the schema integration field. The focus is on the concepts, the proposed solutions, not on detailed technical discussions. We identify two major issues in schema integration:

- Interschema correspondence;
- Interschema conflict.

9.3.1 Interschema Correspondences

Interschema correspondences are frequently found by looking for similarities in the input schemas. Two local schemas are said to have *something in common* if the real world subsets they represent have some common elements (i.e., a non-empty intersection) or have some elements related to each other in a way that is of interest for future applications [Parent, 2000]. At the instance level, two constructs (occurrence, value, tuple, link,...) from two databases are said to correspond to each other if they describe the same world element (object, link or property).

If a correspondence can be defined such that it holds for every construct in an identifiable set (e.g., the population of a type), the correspondence is stated at the schema level. This definition of a correspondence is called an *Interdatabase Correspondence Assertion* (ICA). The complete integration of existing databases requires an exhaustive identification and processing of all relevant ICA.

In an exhaustive approach, the integration process consists in finding all interschema correspondences and for each correspondence adding to the integrated schema and an integrating description of the related elements (supporting the mapping at the instance level). Local elements with no counterpart are directly integrated in the global schema. At the end of the process, the integrated schema provides a complete and non-redundant description of all data in the global schema. The mappings between the integrated schema and the local schemas support integrated data access for users of the global schema.

Classification and detection

The objective of this phase is to identify constructs in the local schemas that may be related and to classify the relationships among them. It is a two-phases process consisting of:

- identifying constructs that are related;
- classifying the relationships among constructs.

The first phase requires that the intended semantics of constructs in databases be extracted and constructs that are semantically related be identified. Once a potential set of related constructs has been identified, the second phase involves classifying these relationships into various categories.

Roughly speaking, two categories of approaches are distinguished: approaches based on a structural analysis and approaches based on an instance analysis.

Approaches based on a structural analysis. Approaches based on a structural analysis use the knowledge conveyed by the various schematic constructs to deduce relationships among constructs. Entity types, attributes and relationships represent the primary schematic constructs that can be analyzed to arrive at these relationships. [Larson, 1989] uses various properties of an attribute to establish relationships among attributes of two different entities belonging to different schemas. The author suggests that attributes be compared on their properties, and it provides definitions for assessing the degree of equivalence of the attributes. For example, entity type could be compared on their names and the description of their roles in the schema. To support the comparison on names, role, etc., sophisticated dictionary and thesauri could be used.

The objective of analyzing these schemas is to identify constructs that are semantically related. However, it is necessary not only to identify but also to classify the relationships among these constructs. The classification generated is dependent on the methodology used. For instance, [Larson, 1989] generates four types of equivalences between attributes. There are a EQUAL b, a CONTAINS b, a CONTAINED-IN b, a OVERLAP b. It goes on to define five types of relationships among entities and relationships, each of which can be derived based on attribute equivalences of key attributes. These relationships include A EQUAL B, A CONTAINS B, A CONTAINED-IN B, A OVERLAP B and A DISJOINT B. Users are asked to specify one of these types of relationships for every entity/relationship whose attributes have equivalence relationships specified on them.

Approaches based on an instance analysis. The objective of these approaches is to determine instances of entity types in different sources that refer to the same real-world entity. The simplest approach assumes that relations from different databases have a common key. Hence, types that have a common key value identify the same real-world entity [DeMichiel, 1989]. However, a common key may not always be available. This is referred to as the *key equivalence problem*. More details can be found in [Elmagarmid, 1999] and [Ramesh, 1997].

Resolution

As mentioned before, schema integration consists in determining the interschema correspondence assertions (ICA) by considering both the instance and structural levels. At the instance level, the entity types of the local schemas are usually set into a relationship with respect to the *extensional assertions* [Spaccapietra, 1991]. According to [Spaccapietra, 1991], the following binary extensional assertions can be specified between two classes: *disjointness* (\neq), *equivalence* (\equiv), *containment* (\supseteq), and *overlap* (\cap). The disjointness assertion states that the two types are extensionally disjoint in each corresponding database state. The equivalence assertion says that the two types are always extensionally equivalent. The containment assertion is used to describe the fact that one type always extensionally contains the other class. The overlap assertions means that the two types can overlap, that is, they may contain types that refer the same real-world objects.

Types of different local schemas are set into a relationship by a set of schema integration operations. The following list, proposed by [Türker, 1999], enumerates the basic operations for "integrating" two entity types ET_1 and ET_2 with the respective intensions¹ I_1 and I_2 and extensions² C_1 and C_2 .

- *Generalization*: the entity types ET_1 and ET_2 are generalized by a new entity type. The intension of the new entity type is determined by the intersection of the intensions of the entity types ET_1 and ET_2 . The extension of the new entity type is given by the union of the extensions of the entity types ET_1 and ET_2 .
- *Specialization*: a new entity type is created as a specialization of the entity types ET_1 and ET_2 . The intension of the new entity type is determined by the union of the intensions of the entity types ET_1 and ET_2 . The extension of the new entity type is given by the intersection of the extensions of the entity types ET_1 and ET_2 .
- *Subtype*: one entity type becomes the subtype (supertype) of the other entity type in the global schema.
- *Merging*: the entity types ET_1 and ET_2 are merged into a new entity type. The intension of the new entity type is determined by the union of the intension of the entity types ET_1 and ET_2 . The extension of the new entity type is given by the union of the extensions of the entity types ET_1 and ET_2 . However, since the new entity type contains more attributes than the input entity types, default or null values have to be generated for some attributes.
- *Partitioning*: the entity types ET_1 and ET_2 are partitioned into entity types with disjoint instance sets. Extensionally, overlapping entity types lead to three entity types ($ET_1 \setminus ET_2$), ($ET_1 \cap ET_2$), ($ET_2 \setminus ET_1$) in the global schema. The entity type ($ET_1 \setminus ET_2$) contains all the structures of the entity type ET_1 that are not in ET_2 . Analogously, the entity

1. The intension of an entity type is determined by the set of structure definitions.

2. The extension of an entity type refers to an actual state of the database at a given time.

type $(ET_2 \setminus ET_1)$ refers to all structures of the entity type ET_2 that are not in ET_1 . The entity type $(ET_1 \cap ET_2)$ comprises all structures that are in both entity types. The intension of the entity type $(ET_1 \setminus ET_2)$ equals the intension of the entity type ET_1 . Analogously, the intension of the entity type $(ET_2 \setminus ET_1)$ equals the intension of the entity type ET_2 . The intension of the entity type $(ET_1 \cap ET_2)$ contains the union of the intensions of both entity types.

The basic schema integration operations are illustrated in Figure 9-6.

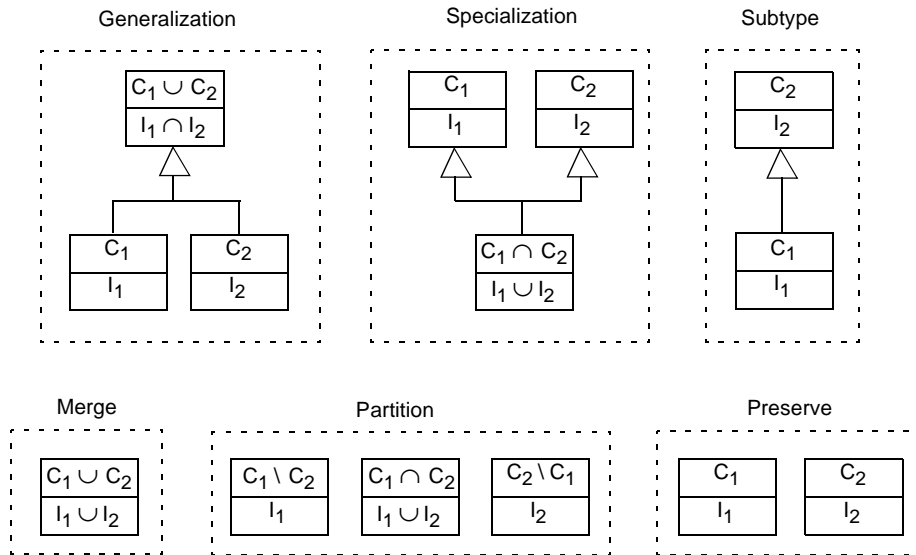


Figure 9-6: Schema integration strategies (from [Türker, 1999]).

9.3.2 Interschema Conflicts

When an ICA describes the corresponding types as identical, integration is straightforward. Most frequently, however, the corresponding types can have some discrepancies at both the structural and instance levels. This situation is called a *conflict*.

A detailed taxonomy of conflicts can be found in [Sheth, 1993b]. In this section, we classify the most common conflicts. Starting with structural conflicts for an overall view, we relate them to instance conflicts and discuss basic techniques for conflict detection.

Structural conflicts

Due to heterogeneity at the schema level, schema integration has to deal with various kinds

of conflicts. A basic classification of structural conflicts has been introduced in [Spaccapietra, 1991]. Following this classification, structural conflicts are classified according two main classes:

- description conflicts;
- semantic conflicts.

Description conflicts. The class of description conflicts comprise a large number of more specific conflicts. Here, we only give some examples that illustrate typical description conflicts.

Corresponding entity types are often described by different attributes in the local schemas. This is due to different requirements of the local applications. In one legacy system, local applications need a certain attribute of the entity type whereas in another legacy system, no application requires this attribute. This refers to the *missing attribute* in similar entity types in different schemas [Elmagarmid, 1999]. The solution is to define an abstract type that contains an aggregation of attributes from the underlying types and returns null for those instances that originate from the type that has the missing attribute. However, the query that explicitly retrieves those instances that have null value for the attribute must not retrieve instances from the type that has the missing attribute, since the attribute itself is undefined rather than the values being unavailable.

Other often occurring description conflicts result from the usage of homonyms and synonyms for attribute names, entity type names, etc. In general, homonyms and synonyms cannot be resolved in a fully automated way.

Further examples for description conflicts are that corresponding attributes may have different data types or ranges in different legacy systems. Even if they have the same data type, different units of measurement or a different scaling can be using within the legacy systems.

Semantics conflicts. This class of conflicts is caused by the usage of different modeling concepts for expressing the same real-world fact. All data models offer several possibilities to model the same real-world fact. Thereby, local schemas expressed in the same data model can have different structures and constraints although they describe the same real-words facts.

Semantics conflict detection requires knowledge about the problem domain, the local schemas and the extensional correspondences. This task can be supported by thesauri or ontologies, but in general an automatic detection can only succeed in very restricted cases or application domains.

Instance conflicts

This type of conflict occurs at the instance level if corresponding occurrences have conflicting values for corresponding structures. For instance, the same order is stored in two different databases with different customer identification values. Sources for instance conflicts include typing errors, variety of information providers, different versioning, deferred updates [Parent, 2000]. These conflicts are normally found during query processing. The system may just report the conflicts to the user, or might apply some heuristic to determine the appropriate val-

ue.

[Sattler, 2002] identifies three kinds of instance conflicts; namely, representation conflicts, identifier equivalence conflicts and attribute value conflicts. For data models with richer expressive power, we could add a further conflict class which refers to relationship conflicts [Lim, 1998].

Representation conflicts. This refers to different representation of data values corresponding to the same real-world fact. This could be caused, e.g., by different unit of measurements (e.g., Belgian Franc vs. Euro), by different notations (e.g., "firstname lastname" vs. "lastname, firstname") or simply different representations (e.g., ISBN with dashes vs. without dashes).

Identifier equivalence conflicts. These arise when instances from different entity types refer to the same real-world object but contain different identifiers.

Attribute value conflicts. They occur when instances, which correspond to the same real-world type and share an equivalent identifier, differ in other attributes. One reason for this problem could be a situation, where two entity types from different sources overlap semantically and one of the entity type contains older or outdated data.

9.4 InterDB Approach

The InterDB approach does not impose strict guidelines to integrate schemas. Experience has shown that this process must be coped with through very flexible techniques, and that different problems in the same federation may require different techniques. In addition, steps generally addressed in theoretical approaches to schema integration are of a lesser importance in the InterDB framework since they have been performed in the reverse engineering process. This is the case for conflict identification and conflict resolution.

Indeed, the reverse-engineering process (Chapter 6) has given analysts a strong knowledge of the semantics of each construct of the local conceptual schemas. In addition, the normalization step should have produced fairly neutral schemas, in which few complex representation conflicts should remain. Therefore, identifying similar constructs and merging them is much easier than when one processes still unidentified logical schemas as proposed in most federated schema building methodologies.

9.4.1 InterDB Principles

Conflict identification

In the InterDB integration methodology, conflicts can occur in three possible ways: syntactic,

semantic and instance.

Syntactic conflict. Besides the usual conflicts related to synonyms and homonyms, a syntactic conflict occurs when the same concept is presented by different object types in local schemas.

Example

For instance, the concept `OrderDetail` can be represented by an attribute (Figure 9-7, site 1) or by an entity type (Figure 9-7, site 2).

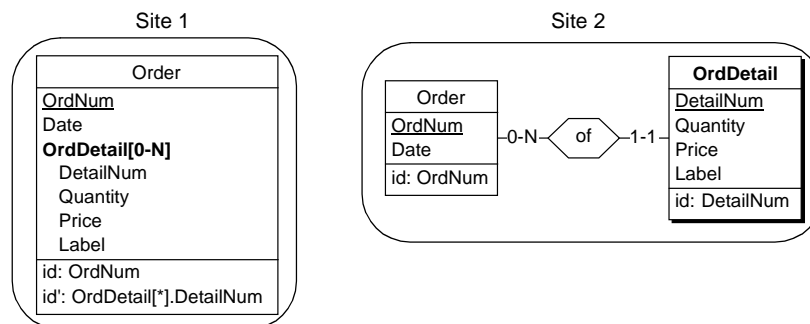


Figure 9-7: Syntactic conflict examples. The concept `OrdDetail` is a multivalued attribute in the site 1 whereas it is represented by an entity type in the site 2.

Semantic conflict. A *semantic conflict* appears when a contradiction appears between two representations A and B of the same application domain concept or between two integrity constraints. Solving such conflicts uses reconciliation techniques, generally based on the identification of set-theoretic relationships between these representations: $A = B$, $A \text{ in } B$, $A \text{ and } B \text{ in } AB$, etc.

Example

Figure 9-8 shows an example of a semantic conflict between two attributes. In the site 1, the attribute `Phone` is optional (its minimal cardinality is 0) whereas the same attribute is mandatory in the site 2.

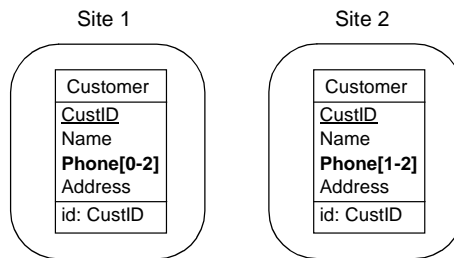


Figure 9-8: Semantic conflict example. The attribute Phone is optional in the site 1 and mandatory in the site 2.

Instance conflicts. Instance conflicts are specific to existing data. Though their schemas agree, the instances of the databases may conflict. This problem has been discussed in [Vermeer, 1996]. This process is highly knowledge-based and cannot be performed automatically.

Example

Consider the two local schemas of the top part of Figure 9-9. Common knowledge suggests that USER be a subtype of EMPLOYEE. However, data analysis shows that $\text{inst}(\text{EMPLOYEE}) \subseteq \text{inst}(\text{USER})$, where $\text{inst}(A)$ denotes the set of instances of data type A.

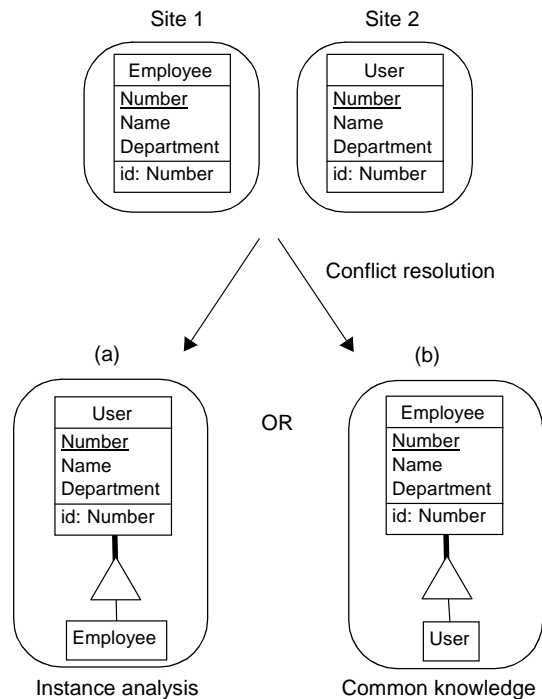


Figure 9-9: Instance conflict example.

Conflict resolution

In the InterDB project, we state that most conflicts can be solved through four main techniques that are used to rework the local schemas before their integration: renaming, generalizing, transforming and discarding.

1. *Renaming.* Constructs that denote the same application domain concepts are given the same name.
2. *Generalizing.* If two constructs denote the same application domain concept, and if one of them is more constrained, the constraint is relaxed. For example, a [0-10] cardinality conflicts with a [1-N] cardinality. Both will be replaced with cardinality [0-N], which is the strongest constraint compatible with both source cardinalities.
3. *Transforming.* An application domain concept can be represented by constructs of different nature in source schemas. A supplier can be represented by an entity type in schema 1 and by an attribute in schema 2. The latter construct will be transformed into an entity type to give both representations the same nature.

4. *Discarding.* A construct that conflicts with others can be merely ignored. This is the case when the former appears to be a wrong translation of the application domain concept.

LCS merging

Since the syntactic, semantic and instance conflicts have been resolved by restructuring the local schemas, merging the latter is fairly straightforward, and can be automated to a large extent. Note that conflict resolution need not be completed as a preliminary process. Indeed, conflicts can be completely or partially solved when merging schemas. According to this strategy, the source schemas are left unchanged, and merging each pair of (sets of) constructs can imply on-the-fly restructuring in order to solve conflicts.

9.4.2 Practical InterDB Methodologies

The InterDB approach recommends flexible and adaptive procedures, that are supported by the DB-MAIN CASE tool, as will be shown in Chapter 10. Two main complementary strategies are proposed. They will be described as scenarios for integrating two schemas, though they can be generalized to N-ary strategies. In actual situations, both strategies can be used alternately to solve different parts of the integration work.

Synthetic strategy

This procedure is proposed for situations in which semantically similar parts of the schemas have almost identical representations. It is based on the following denotation assumptions:

- two objects of the same nature (entity type, relationship type or attribute) with the same name denote exactly the same application domain concept,
- any pair of objects that does not satisfy this condition denote independent application domain concepts.

This traditional strategy includes two sequential steps.

1. *Pre-integration.* This step is intended to make both schemas satisfy the denotation assumptions. Similar objects are identified and, if needed, their name and nature are modified accordingly. New objects can be introduced. For instance, if entity type E_2 in schema 2 is recognized as a subtype of E_1 in schema 1, then an empty entity type with name E_1 is created in schema 2, and made a supertype of E_2 .
2. *Global merging.* The schemas are merged according to the denotation assumptions. It is based on the following rules:
 - if two entity types have the same name, they are merged, i.e., only one is kept, and their attributes are merged; non matching attributes of both entity types are kept;
 - if two attributes of merged objects have the same name, they are merged, i.e., only one is kept, and their attributes, if any, are merged; non matching attributes

of both parent objects are kept;

- if two relationship types have the same name, they are merged, i.e., only one is kept and their roles and attributes are merged; non matching roles and attributes of both relationship types are kept;
- if two roles of merged relationship types have the same name, they are merged, i.e., only one is kept, and their attributes, if any, are merged.

This leads to a straightforward algorithm that can be easily automated.

Analytical strategy

The second strategy will be used in more complex situations. It consists in integrating pairs of constructs individually.

1. *Identifying similar constructs and their semantic relation.* The process is based on the knowledge gained by the analyst during the reverse engineering process, and on similarities between related parts on the source schemas (such as name and structural closeness). The semantic relation is identified. We suggest to choose one of the following five situations:
 - identity: the constructs denote the same concept;
 - complementarity: the constructs represent two facets of the same concept;
 - subtyping: one construct denotes a subclass of the concept denoted by the other one;
 - common supertype: both constructs denote subclasses of an implicit concept;
 - independence: the constructs denote independent concepts.
2. *Solving representation conflicts.* If necessary, names are changed and transformations are applied to make merging in step 3 easier.
3. *Merging.* We consider the typical binary strategy in which the master schema is enriched from the contents of a slave schema, which remains unchanged. According to the five situations identified in step 1, applied to constructs M in the master schema, and S in the slave schema, six actions will be proposed.
 - identical(M,S): the components of S are transferred to M;
 - complementarity(M,S): a copy of S is created in the master schema and is linked to M;
 - subtype_of(M,S): a copy of S is created in the master schema and is made a subtype of M;
 - subtype_of(S,M): a copy of S is created in the master schema and is made a supertype of M;
 - common_supertype(M,S): a copy of S is created in the master schema and a new construct is created and made the common supertype of M and S;
 - independent(M,S): if the relation is true for all M's, a copy of S is created in the

master schema.

To make things more complex, the process must be considered recursively. Indeed, each construct generally has components: an entity type has a name, attributes, roles, constraints and textual annotations; an attribute has a name, a type, a length, sub-attributes and textual annotations; a relationship type has a name, roles, attributes and textual annotations; a role has a name, cardinality, one or several participating entity types and textual annotations.

In each merging technique (but the last one) the components of *M* and *S* must be compared pairwise, to identify their semantic relation and to decide on their integration strategy. For instance, considering attribute *AS* of *S*, either *AS* is identical to attribute *AM* of *M*, in which case they will be merged, or *AS* must be added to *M*. In the former case, *C* components (name, type, annotation, etc.) of *AS* and *AM* are compared pairwise. Either they match, in which case *AM.C* is kept, or they conflict, in which case a human decision must be made: either *AM.C* or *AS.C* is kept, or a combination of both is adopted as *AM.C* (e.g., the concatenation of the annotations).

Building the inter-schema mappings

As stated in Chapter 4, our approach is schema transformation oriented in that we focus on providing mechanisms for defining schema correspondences between each local schemas and the global schema (Figure 9-10), and, on then using that equivalences to defined the mediator mappings. As a result, transformation sequences for each pair of <local schema, global schema> have to be defined.

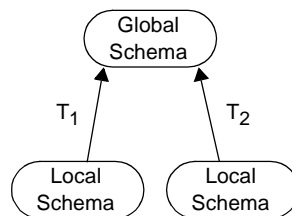


Figure 9-10: Schema integration as a set of schema transformation sequences.

Mediator Development Support

In which we analyse the requirements that CASE tools should meet for the development of mediator systems. We then present an operational CASE tool - DB-MAIN - which is intended to address some of these requirements.

10.1 Introduction

The mediator development methodology presented in Chapter 9 describes the general principles that can be used to achieve schema integration and mediator definition. It is clear from this discussion that mediator development and schema integration are complex and time-consuming processes, and automation is desirable. However, automation of the process presents number of challenges. [Sheth, 1989] notes that the schema integration process cannot be completely automated. This is primarily because two same schemas can be integrated differently based on their intended use [Sheth, 1991]. It is however possible to reduce the amount of human interaction [Elmagarmid, 1999].

No tool has yet been developed as a commercial product. Some research projects have produced significant prototypes. They are dedicated either to the integration process or to the building of mediators:

- *For the integration process:* [Hayne, 1992], [Gotthard, 1992] and [Ramesh, 1995] propose tools for automated interschema relationship identification. [Schwarz, 1998] presents a set of tools that support different issues of the process, e.g., methodology, conflicts and similarities identification, semantics extraction.

- *For the building of federation components:* [Papakonstantinou, 1995] proposes an implementation toolkit that facilitates the rapid development of mediators; [Vidal, 1998] presents a meta-mediator providing a single meta-mediator interface for all the sources. HERMES [Subrahmanian, 1995] provides a set of tools to support the construction of mediators.

Many of these tools, however, appear to be limited in scope, and are generally dedicated to a limited aspect of the federation development. They do not attempt to integrate techniques and reasoning common to the integration process and the building of federation components, leaving the question of a general tool for developing a mediator unanswered.

10.2 CASE Tool Requirements

The requirements of CASE tools dedicated to the mediator development are quite similar to those dedicated to the wrapper development (Chapter 7). In this section, we study some problems specific of the mediation.

Support of the integration

Several integration strategies can be applied, depending on the complexity and the heterogeneity of the source databases and on the skill of the analyst. However, there exists a collection of commonly used conflicts strategies that can be applied for conflicts solving [Subrahmanian, 1995]. The tool must include a collection of basic techniques for the integration instead of a unique, automated, schema integrator. It must include a set of predefined functions for detecting conflicts.

Each federation is a new problem of its own, requiring specific reasoning and techniques. Integrating local schemas appears as a learning activity. The predefined tools should be easy to customize and to program, and, specific functions should be easy to develop.

Solving syntactic, semantic and instance conflicts requires a great variety of information sources: schemas, data (files, databases, spreadsheets, etc.), data mining analysis, domain knowledge, etc. Hence, the tool must include browsing and querying interfaces with these sources. Customizable functions for assisted specification analysis should be available for each of them. In particular, the tool must include data mining techniques for the instance conflict identification.

Support of the mapping definition

Mediator specifications are based on mapping definition. The CASE tool should automatically generate and maintain information about mappings between schemas. It must also include sophisticated automatic or assisted mapping analyzers and provide several ways of viewing both mapping definition and schemas.

Moreover, further information (e.g., transaction management or security) is necessary to build efficient mediators. The CASE tool should maintain any type of information that can be used for specific need.

10.3 DB-MAIN

The DB-MAIN CASE environment [Hick, 2002] is a complete set of tools dedicated to database applications engineering. This graphical, repository-based, software engineering environment is dedicated to database applications engineering. The DB-MAIN CASE tool addresses the main requirements developed in the previous sections. As a large-scope CASE tool, DB-MAIN includes usual functions needed in data analysis and design, e.g. entry, browsing, management, validation, transformation, as well as code and report generation.

This graphical, repository-based, software engineering environment is dedicated to database applications engineering. Besides standard functions such as specification entry, examination and management, it includes advanced processors such as transformation toolboxes, reverse engineering processors and schema analysis tools.

It also provides powerful assistants to help developers and analysts carry out complex and tedious tasks in a reliable way. The assistants offer scripting facilities through which method fragments can be developed and reused.

One of the main features of DB-MAIN is the Meta-CASE layer, which allows method engineers to customize the tool and to add new concepts, functions, models and even new methods. The InterDB project has customized the DB-MAIN tool by extending its repository and by adding concepts and processors specific to the mediator development.

10.3.1 DB-MAIN Repository

The repository collects and maintains all the information related to a schema integration. The repository comprises three classes of information:

- a structured collection of schemas and texts used and produced in the wrapper development,
- the specification of the methodology followed to conduct the wrapper development,
- the history (or trace) of the schema integration.

We will ignore the second class, which are related to methodological control and which is described in [Roland, 1997]. We will discuss the third class in Section 10.3.3.

A *schema* is a description of the data structures to be processed, while a text is any textual material generated or analyzed during the project (e.g. a program or an SQL script). A mediator development comprises several schemas. The federation schemas are linked through specific relationships. The schema specification is based on the generic model defined in Chapter

3. Besides the standard concept of the generic model, the repository includes some meta-objects which can be customized according to specific needs. In addition, annotations can be associated with each object. These annotations can include semi-formal properties, made of the property name and its value. These features provide dynamic extensibility of the repository. For instance, new concepts such as mapping definition can be represented by specializing the meta-objects, while statistics about entity populations can be represented by semi-formal attributes.

10.3.2 Integration Assistants

Schema integration occurs mainly when merging the local conceptual schemas into the global schema. It also appears in reverse engineering to merge multiple descriptions into a unique logical schema. In addition, several strategies can be applied, depending on the complexity and the heterogeneity of the source databases and on the skill of the analyst. As a consequence, DB-MAIN offers a toolbox for schema integration instead of a unique, automated, schema integrator. Together with the transformation toolbox, the integration toolbox allows manual, semi-automatic and fully automatic integration. The synthetic strategy is supported by a schema integration processor that is based on the denotation assumptions. The analytical strategy uses different processors, namely, the schema integration assistant and the object integration assistant.

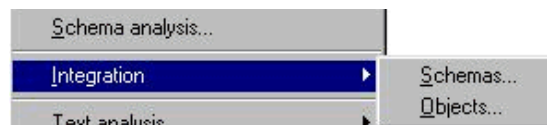


Figure 10-1: Schema and object integration.

Schema integration assistant

This assistant integrates a schema into another schema (Figure 10-2) by using predefined rules.



Figure 10-2: Schema integration assistant.

The rules used to integrate a (slave) schema into the another one (the master) are:

- If the slave data schema contains a new entity type, it is created. If the entity type already exists, see the rules for two entity types with the same name.
- If the slave data schema contains a new rel-type, it is created. If the rel-type already exists, see the rules for two rel-types with the same name.
- If the slave data schema contains a new collection, it is created. If the collection already exists, see the rules for two collections with the same name.
- *Two entity types with the same name:* the short name is not modified. If there is an is-a relation in the slave schema, the connection is created to the cluster if the connection does not exist. If the entity type in the slave schema contains a new attribute, it is created. If the attribute already exists, see the rules for two attributes with the same name. If the entity type in the slave schema contains a new group, it is created. If the group already exists, see the rules for two groups with the same name.
- *Two rel-types with the same name:* the short name is not modified. If the rel-type in the slave schema contains a new attribute, it is created. If the attribute already exists, see the rules for two attributes with the same name. If the rel-type in the slave schema contains a new role, it is created. If the role already exists, see the rules for two roles with the same name. If the rel-type in the slave schema contains a new group, it is created. If the group already exists, see the rules for two groups with the same name.
- *Two roles with the same name:* the short name and the cardinality are not modified. If, in the slave schema, the role is connected to an entity type to which it is not connected in the schema, then the connection is created.

- *Two attributes with the same name:* the cardinality and the short name are not modified. If the master is a not compound attribute and the slave is a compound attribute, the master attribute is deleted and replaced by the slave one. If the master is a compound attribute and the slave not, the master is not modified. If they are both compound or not, the master is not modified. If the attribute in the slave schema is a compound attribute that contains a new attribute, it is created. If the attribute already exists, see the rules for two attributes with the same name. If the attribute in the slave schema contains a new group, it is created. If the group already exists, see the rules for two groups with the same name.
- *Two groups with the same name:* add the components that are defined in the slave schema to the group if they are not present in the master. If, in the slave schema, the group is the origin of a constraint, this constraint is added and the other one in the master (if it exists) is deleted.
- *Two collections with the same name:* Short name is not modified. Add to the collection the entity type that were not connected.

Object integration assistant

The *object integration tool* (Figure 10-3) integrates two objects (entity types, relationship types or compound attributes) in the same data schema or between two different schemas (from the slave to the master). There are six integration strategies. Attributes, processing units, roles, is-a relations and their properties can be migrated selectively.

The *object matching dialog box* (Figure 10-4) is called by the same button and compares two different components (attributes, processing units, roles or is-a relations) of the master and slave objects.

Example

Figure 10-3 shows the integration of entity types ORDER and ORD. When asserting that ORDER.OrdNumber and ORD.Ord-Code are the same (Figure 10-3, button Same), the assistant compares their properties and presents them whenever a conflict is detected. Solving this conflict is up to the analyst (Figure 10-4).

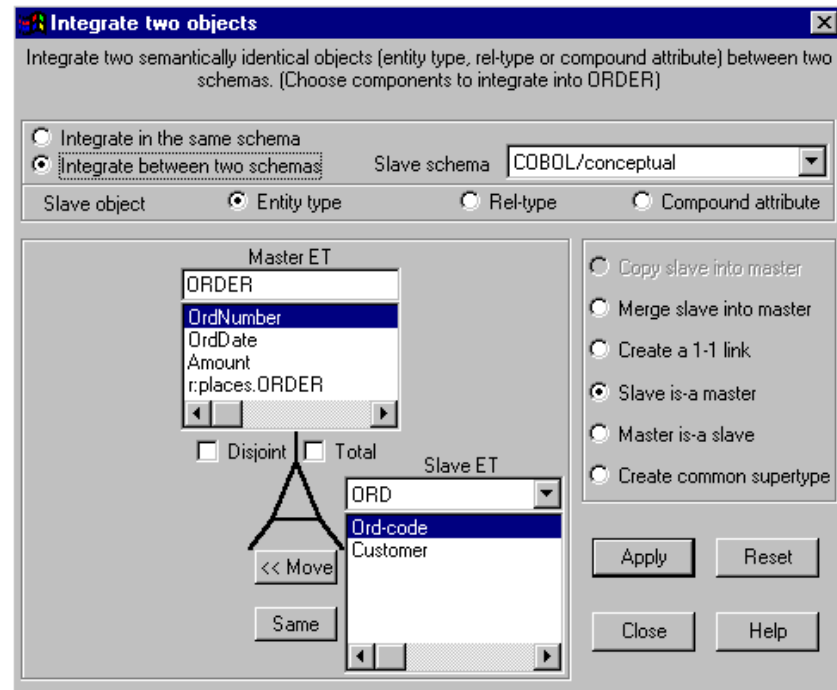


Figure 10-3: The integration assistant. Entity types ORDER and ORD of two distinct local schemas are examined for integration. Among the six integration strategies, the analyst chose the fourth one, according to which ORDER is a supertype for ORD. The attributes and roles are compared and either migrated (button <<Move) or merged (button Same). Here, the analyst is going to tell that attributes OrdNumber and OrdCode have the same semantics, and that only the first one must be kept.

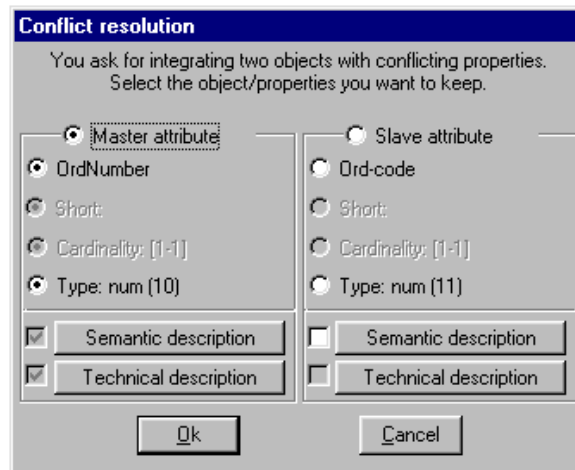


Figure 10-4: The integration assistant: resolving the conflicting properties of attributes OrdNumber and OrdCode that have been declared to be the same. Three conflicting properties have been identified: name, type and semantic description.

10.3.3 History

DB-MAIN automatically generates and maintains a history log (say h) of all the activities that are carried out when the developer derives a schema B from schema A . This history is completely formalized in such a way that it can be replayed, analyzed and transformed. For example, any history h can be inverted into history h' . Histories must be normalized to remove useless sequences and dead-end exploratory branches.

If h_i expresses the structural mapping between a local schema LS_i and the global schema GS , and if t_i is the instance mapping of h_i , then $\{h_i', t_i\}$ is the functional specification of the mapping pair $\langle LS_i, GS \rangle$. For the global query decomposition into local queries, the mediator relies on the set of pairs $\langle LS_i, GS \rangle$ defined on each local schema and the global schema (Figure 10-5).

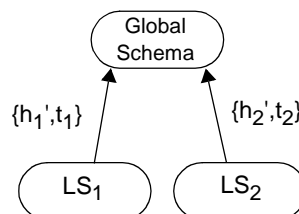


Figure 10-5: Histories between each pair of \langle local schema, global schema \rangle .

10.4 InterDB Tools

The mediator is based on an extension of the DB-MAIN repository that describes the federated (or global) object-oriented schema, the local object-oriented schema of each wrapper, its location, and the relationships between local and global schemas.

Two tools have been built around the DB-MAIN repository:

- *History analyzer.* A history basically is a procedural description of inter-schema mappings. The history analyzer analyses the histories in order to transform them into functional specifications from which the global schema is enriched with its correspondences. At current time, the history analyzer is not yet developed.
- *JiDBM interface.* The JiDBM interface is a Java API for accessing to the DB-MAIN repository. JiDBM gives the Java-written mediator the access to the DB-MAIN repository.

The interaction between these tools and the DB-MAIN repository is illustrated in Figure 10-6.

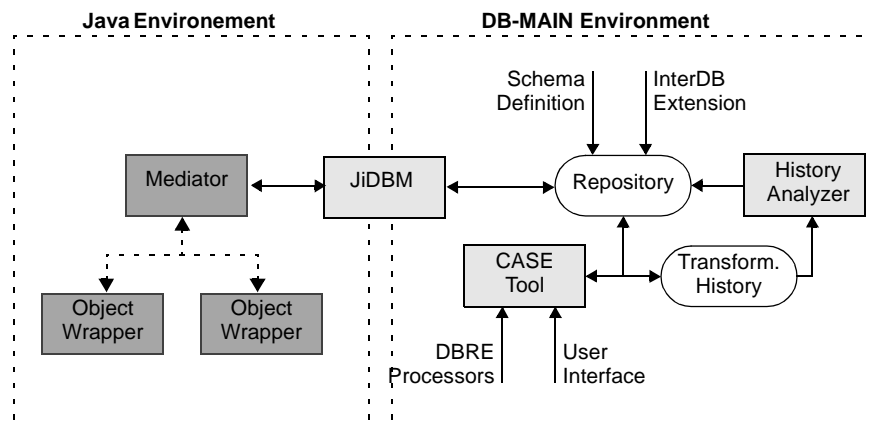


Figure 10-6: The DB-MAIN tool and its interface with the mediator.

10.4.1 InterDB Extension of the DB-MAIN Repository

The DB-MAIN repository has been extended so that it is able to collect and maintain all the information related to a database federation. The repository comprises three classes of information:

- Schema description;
- User-defined definition;
- Mediation operators (selection, union or join).

Schema description

A schema is a description of the data structures to be processed. A database federation usually comprises many (i.e. dozens to hundreds of) schemas. The schemas of a database federation are linked through specific relationships; they pertain to the federation hierarchy structure.

The DB-MAIN repository has been extended with meta-properties that represent the global/local correspondences. A meta-property is defined as a triple $\langle \text{name}, \text{construct}, \text{value domain} \rangle$ that specifies that a construct is associated with a meta-property of value domain value domain. Some of these meta-properties are represented in Figure 10-7.

Meta Property	Construct	Value
InterDB-operator	Object type	selection union join
InterDB-local-sources	Object type	list of the sources
InterDB-local-objects	Object type	list of the local object types
InterDB-matching	Object type	list of the comparison local attributes
InterDB-function	Object type	name of the reconciliation function
InterDB-local-attributes	Attribute	list of the local corresponding attributes
InterDB-function	Attribute	name of the conversion function

Figure 10-7: Meta-properties defining the mapping properties between the local and global object schemas.

User-defined function

For resolving domain discrepancies, *user-defined functions* are introduced. A typical application is the conversion of attribute values (for instance, the Euro conversion of the Belgian Franc) which are represented in a legacy database in a different way than needed by the actual applications.

As introduced in the previous section, the schema transformations are defined as functional extensions of the object types. A user-defined function can be attached to a schema transformation of a domain. Such a function is therefore used to transform attribute values if the function is attached to an attribute. We define the user-defined functions as:

Definition. Let us consider an attribute A of a global object type OT : $f : t_A \rightarrow t$ is a function which can be applied to values of domain t_A defined for the attribute A of OT resulting in values of domain t .

Example

As an example for user-defined function, consider the conversion function for a price attribute which converts Belgian Franc to Euro (Figure 10-8).

```
Integer BelgianFranc2Euro(Integer price) {
    Integer priceEuro = new Integer();
    priceEuro = price / 40,3399;
    return priceEuro;
}
```

Figure 10-8: User-defined function example.

Mediation operators

For resolving instance discrepancies or conflicts, three mediation functions are introduced: the *selection*, *join* and *union* operators.

Selection function. The *selection function* is defined for resolving instance conflicts in a same object by selecting and applying a reconciliation function. A reconciliation function is a *user-defined function* which is called for each instances fulfilling the selection condition. The affected instances are passed as arguments to the function, the resulting instance leads to the object result.

Definition. Let us consider a global object type OT which is defined by the selection of an entity types OT_1 . OT and OT_1 belong to different schemas. Let us consider α as the selection condition and f as the reconciliation function. The general form of this selection function is: $OT \leftarrow selection(OT_1, \alpha, f)$

The function f is mandatory. The selection function is considered to be the selection product of OT_1 using the selection condition α . In summary, the join operation with a reconciliation function f can be represented as follows: $OT \leftarrow f(\alpha_{\alpha}(OT_1))$ where α_{α} is the selection function defined by the selection condition α .

Join function. The *join function* is defined for integrating two local objects by joining and applying a reconciliation function for resolving possible conflicts between certain attributes. A reconciliation function is a *user-defined function* which is called for each instances fulfilling the comparison condition. The affected instances are passed as arguments to the function, the resulting instance is inserted into the global object. In this way, the value of a global attribute can be computed from the (possibly) conflicting values of the corresponding local attributes.

Definition. Let us consider a global object type OT which is defined by the join of two local object types OT_1 and OT_2 . OT_1 and OT_2 can belong to different schemas. Let us consider α as the join condition and f as the reconciliation function. The general form of this join function is: $OT \leftarrow join(OT_1, OT_2, \alpha, f)$.

The function f is optional. Without a reconciliation function, the join function is considered to be a Cartesian product of OT_1 and OT_2 with a subsequent selection using the selection condition α .

In summary, the join operation with a reconciliation function f can be represented as follows: $OT \leftarrow f(\alpha_{\alpha}(OT_1 \times OT_2))$ where α_{α} is the selection function defined by the join condition α .

Example

Consider the local object types in Figure 10-9. WOS_1 holds information about customers whereas WOS_2 holds information about their orders. In WOS_2 , the object type Customer has an attribute `CustID` that references the customers recorded in WOS_1 .

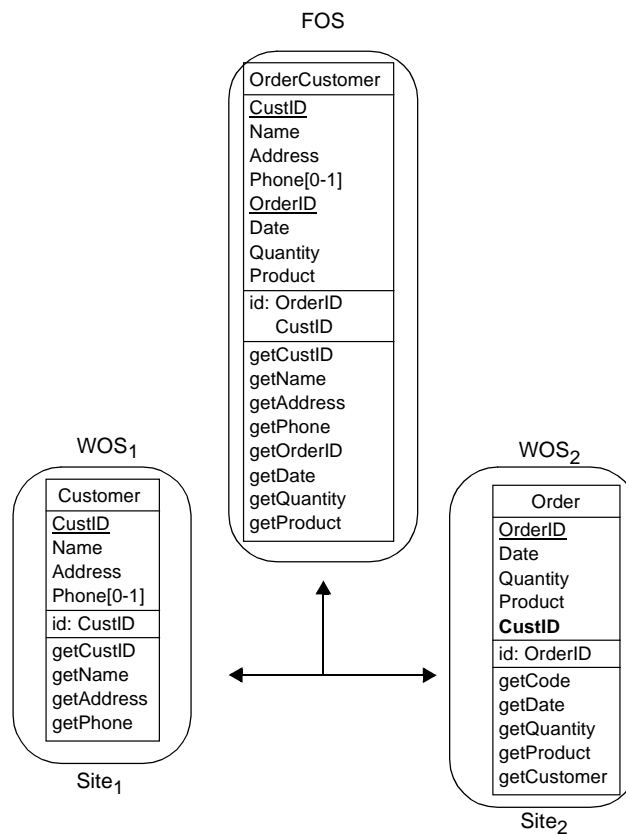


Figure 10-9: Global object defined as the join of two local objects.

The parameters of the join operator are translated as meta-properties of the global object type. Its meta-properties for the global object type `OrderCustomer` are given in Figure 10-10.

FOS.Product (object type)	
Operator	Join
Local objects	WOS ₁ .Customer WOS ₂ .Order
Matching attributes	WOS ₁ .Customer.CustID WOS ₂ .Order.CustID
Reconciliation function	null

Figure 10-10: Meta-property examples.

Union function. The *union function* is another way for integrating two objects by merging them.

Definition. Let us consider a global object type OT which is defined by the union of two local object types OT₁ and OT₂. OT₁ and OT₂ are of the same type. Let us consider α as the union condition and f as the reconciliation function. The general form of this join function is: $OT \leftarrow union(OT_1, OT_2, \alpha, f)$

f is a reconciliation function which takes an instance from the union product of OT₁ and OT₂ and produces an instance of the type OT.

We restrict the type OT in a way that it is the common super type being compatible to both types OT₁ and OT₂. OT has exactly the same attributes (names) and each attribute has the same type.

A *union function* can in principle be computed in the following way:

- by means of a standard union (in the set-theoretic sense, i.e., eliminating duplicates);
- by applying the reconciliation function to each instance produced in the preceding step we compute from the pair of values for each attribute one resolved value. For the attributes in the union condition, the resolved value should be the common value or the value which is different from the NULL value.

Example

To illustrate the union function, let us take up the example of Chapter 8. This example is also showed in Figure 10-11. We recall that the example presents a global object type (FOS) containing product information. This object type is the result of the union of two local object types (WOS₁ and WOS₂) using the product number (Npro) as the matching attribute and a reconciliation function resolveConflict (Figure 10-13) for resolving conflicts between two local objects Product having the same product number but two different prices.

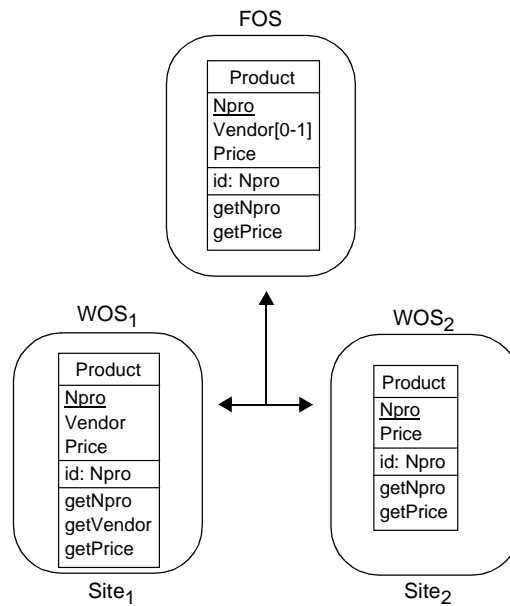


Figure 10-11: Global object type defined as the union of two local object types.

The parameters of the union operator are translated as meta-properties of the global object type. Its meta-properties for the global object type **Product** are given in Figure 10-12.

FOS.Product (object type)	
Operator	Union
Local objects	WOS ₁ .Product WOS ₂ .Product
Matching attributes	WOS ₁ .Product.Npro WOS ₂ .Product.Npro
Reconciliation function	resolveConflict(WOS ₁ .Product, WOS ₂ .Product)

Figure 10-12: Meta-property examples.

```

GlobalObject resolveConflict(LocalObject i1, LocalObject i2) {
    GlobalObject res = new GlobalObject();
    // resolve price and vendor attribute:
    if (i1 != null) {res.setString("Vendor", i1.getString("Vendor"));
        res.setFloat("Price", i1.getFloat("Price"));
        res.setString("Npro", i1.getString("Npro")); } // copy npro

    else {res.setString("Vendor", null);
        res.setFloat("Price", i2.getFloat("Price"));
        res.setString("Npro", i2.getString("Npro")); } // copy npro

    return res;
}

```

Figure 10-13: Reconciliation function example.

10.4.2 History Analyzer

DB-MAIN can automatically generate and maintain a *history log* of all the transformations that are applied when the developer carries out a schema integration. This history is completely formalized in such a way that it can be replayed, analyzed and transformed.

A history basically is a procedural description of inter-schema mapping. This form does not provide a good support for reasoning and processing, for which a functional expression is better suited.

A *Voyager II* program - *history analyzer* - analyses history log in order to transform them into functional specifications from which the global schema is enriched with its correspondences. The end product is an enriched global schema that includes, for each construct of a schema, the way it has been mapped onto the constructs of the underlying local schemas.

The type of the schema transformation is very important since they induce the presence of a user-defined function or a mediation function. For instance, a domain transformation defined for an attribute leads to a user-defined function that is used for transforming attribute values. On the other hand, a naming transformation is sufficient by itself. Other illustrative examples are shown in Figure 10-14.

Transformation (sequence) type	Meta property	Constructs
Renaming an attribute	InterDB-local-attributes	Attribute
Transforming a domain of an attribute	InterDB-function	Attribute
Adding an integrity constraint	InterDB-function	Object type

Transformation (sequence) type	Meta property	Constructs
Merging two identical entity types into one entity type	InterDB-operator=UNION InterDB-local-objects InterDB-local-sources InterDB-matching	Object type
Merging two different entity types into one entity type	InterDB-operator=JOIN InterDB-local-objects InterDB-local-sources InterDB-matching	Object type

Figure 10-14: Transformation types and meta-properties. Some examples.

10.4.3 Java Access to the DB-MAIN Repository

JiDBM (Java Interface for DB-MAIN) is a Java API for accessing to the DB-MAIN repository [Hainaut, 1998]. It consists of a set of classes written in Java programming language. JiDBM makes therefore possible to write Java applications accessing the DB-MAIN repository.

Example

Figure 10-15 presents a small java program which displays some statistics about an ER schema.

```
class test
{
    public static void main(String[] args)
    {
        dbmRepository R = new dbmRepository("g:\\dbmain\\lessai.lun");
        dbmSystem dbm = R.getSystem();
        System.out.println("\nProject" );
        System.out.println("-----");
        System.out.println(" Name:" + dbm.name);
        dbmSchema s = dbm.getFirstSchema();

        if (s!=null)
        {
            System.out.println("\nSchema" );
            System.out.println("-----");
            System.out.println(" Name:" + s.name);
            System.out.println(" Version:" + s.version);
            System.out.println("\nRelationships" );
            System.out.println("-----");
            dbmRelType rt = s.getFirstRelType();
            dbmRelType rt_c = rt;
            while (rt!=null)
            {
                System.out.println(" " + rt.name);
                dbmRole ro = rt.getFirstRole();
                dbmRole ro_c = ro;
                while (ro!=null)
                {
```

```

        System.out.println(" " + ro.name+
            "["+ro.minCon+"."+ro.maxCon+"]");
        ro = rt.getNextRole(ro_c);
        ro_c = ro;
    }
    rt = s.getNextRelType(rt_c);
    rt_c = rt;
}
System.out.println("\nEntities and attributes" );
System.out.println("-----");
dbmEntityType et = s.getFirstEntityType();
dbmEntityType et_c = et;
while (et!=null)
{
    System.out.println(" " + et.name);
    dbmAttribute att = et.getFirstAttribute();
    dbmAttribute att_c = att;
    while (att!=null)
    {
        System.out.println(" " + att.name +
            "["+att.minRep+"."+att.maxRep+"]");
        att = et.getNextAttribute(att_c);
        att_c = att;
    }
    et = s.getNextEntityType(et_c);
    et_c = et;
}
}
else {System.out.println("No schema");}
}
}

```

Figure 10-15: Java program example accessing to the DB-MAIN repository.

References

- [Aslan, 1999] G. Aslan, D. McLeod, "Semantic Heterogeneity Resolution in Federated Databases by Metadata Implementaion and stepwise evolution", *The VLDB Journal*, Vol. 8, pp. 120-132, 1999.
- [Atzeni, 1993] P. Atzeni, R. Torlone, "A Metamodel Approach for the Management of Multiple Models and the Translation of Schemas", *Information Systems*, 18(1), pp. 134-143, 1993.
- [Batini, 1986] C. Batini, M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methologies for Database Schema Integration", *ACM Computing Surveys*, 18(4), Dec. 1986, pp. 323-364.
- [Batini, 1992] C. Batini, S. Ceri and S.B. Navathe, "*Conceptual Database Design - An Entity-Relationship Approach*", Benjamin/Cummings, 1992.
- [Bayardo, 1997] R.J. Bayardo *et al.*, "InfoSleuth: Agend-based Semantic Integration of Information in Open and Dynamic Environment", *SIGMOD Record*, 26(2), pp. 195-206, June 1997.
- [Beneventano, 1997] D. Beneventano, S. Bergamaschi, C. Sartori, M. Vincini, "ODB-QOPTIMIZER: a Tool for Semantic Query Optimization in OODB", in *Proc. of Int. Conference on Data Engineering (ICDE'97)*, 1997.
- [Bergamaschi, 2001] S. Bergamaschi, S. Castano, D. Beneventano, M. Vinci, "Retrieving and Integrating Data for Multiple Sources: the MOMIS Approach", *Data and Knowledge Engineering*, 36, 2001.
- [Bouguettaya, 1998] A. Bouguettaya, B. Benetallah, A. Elmagarmid, "*Interconnecting Heterogeneous Information Systems*", Kluwer Academic Publishers, 1998.
- [Brodie, 1995] M. Brodie, M. Stonebraker, "*Migrating Legacy Systems*", Morgan Kaufmann, 1995
- [Busse, 2000] S. Busse, R-D. Kutsche, U. Leser, "Strategies for the Conceptual Design of Federated Information Systems", in *Proceedings of EFIS'00*, pp. 23-32, IOS Press and Infix, 2000.
- [Cali, 2001] A. Cali, D. Calvanese, G. De Giacomo, M. Lenzerini, "Accessing Data Integration Systems through Conceptual Schemas", in *Proceedings of ER'01*, pp. 271-284, LNCS 2224, Springer-Verlag, 2001.
- [Castellanos, 1994] M. Castellanos, T. Kudrass, F. Saltor and M. Garcia-Solaco, "Interdatabase Existence Dependencies: a Metaclass Approach", in *Proc. 3rd. Int. Conf. on Parallel and Distributed Database System*, pp. 213-216, IEEE Computer Society Press, 1994.
- [Catarci, 1993] T. Catarci, M. Lenzerini, "Representing and Using Interschema Knowledge in Cooperative Information Systems", *Journal for Intelligent and Cooperative Information Systems*, 2(4),

- WorldScientific Press, pp.375-399, 1993.
- [Chandra, 1977] A.K. Chandra, P.M. Merlin, "Optimal Implementation of Conjunctive Queries in Relational Databases", in *Proc. 9th Annual ACM Symposium on Theory of Computing*, pp. 77-90, ACM Press, 1977.
- [Chawathe, 1994] S.S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J.D. Ullman, J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources", in *Proceedings of the 10th Meeting of the Information Processing Society of Japan*, pp. 7-18, 1994.
- [Chen, 1998] Y. Chen, W. Benn, "Query Evaluation for Distributed Heterogeneous Relational Databases", in *Proceeding of CoopIS'98*, IEEE Computer Science Press, 1998.
- [Cimitile, 1998] A. Cimitile, U. de Carlini, A. De Lucia, "Incremental Migration Strategies: Data Flow Analysis For Wrapping", in *Proc. of WCRE '98*, IEEE Computer Society Press, pp. 59-68, 1998.
- [Cluet, 1998] S. Cluet, Cl. Delobel, J. Siméon, K. Smaga, "Your Mediators Need Data Conversion!", in *Proceedings of SIGMOD Conference*, pp. 177-188, 1998.
- [Conrad, 1999] S. Conrad, W. Hasselbring, U. Hohenstein, R-D. Kutsche, M. Roantree, G. Saake, F. Saltor, "Engineering Federated Information Systems - Report of EFIS'99 Workshop", *ACM SIGMOD Record*, 28(3), 1999.
- [Date, 1995] C.J. Date, "*An Introduction to Database Systems*", 6th Edition, Addison-Wesley, 1995.
- [Deacon, 1996] A. Deacon, H-J. Sheck, G. Weikum, "Semantics-based Multi-level Transaction Management in Federated Systems" in *Proc. of 9th Conference on Parallel and Distributed Computing Systems*, pp. 759-765, Raleigh, 1996.
- [De Capitani, 1997] S. De Capitani di Vimercati and P. Samariti, "Authorization Specification and Enforcement in Federated Database Systems", *Journal of Computer Society*, 5(2), pp. 155-188, 1997.
- [Delcroix, 2001] C. Delcroix, Ph. Thiran, J-L. Hainaut, "Approche Transformationnelle de la Ré-ingénierie des Données", *Ingénierie des Systèmes d'Information*, Hermes-Sciences, Paris, December 2001.
- [DeMichiel, 1989] L. DeMichiel, "Resolving Database Incompatibility: an Approach to Performing Relational Operations over Mismatched Domains", in *IEEE Transactions on Knowledge and Data Engineering*, 1(4), pp. 484-493, 1989.
- [Denis, 2002] R. Denis, "*Support à la Conception de Wrappers Conceptuels pour Bases de Données*", Mémoire de Graduat en Informatique, HEMES Liège, 2002.
- [Dogac, 1995] A. Dogac and al., "METU Interoperable Database System", *SIGMOD RECORD*, Vol. 24(3), pp. 56-61, 1995.
- [Elmagarmid, 1999] A. Elmagarmid, M. Rusinkiewicz, A. Sheth, "*Management of Heterogeneous and Autonomous Database Systems*", Morgan Kaufmann, 1999.
- [Englebert, 2001] V. Englebert, "*Voyager II Manual*", DB-MAIN Series, Institut d'Informatique, University of Namur, 2001.
- [Gall, 1995] H. Gall, R. Klösch, "Finding Objects in Procedural Programs", in *Proc. of the 2nd IEEE Working Conf. on Reverse Engineering*, Toronto, IEEE Computer Society Press, July 1995.
- [Garcia, 1995] M. Garcia-Solaco, F. Saltor, M. Castellanos, "A Structure Based Schema Integration Methodology", in *Proceedings of the 11th International Conference of Interoperable Database Systems*, IEEE CS Press, pp. 505-512, 1995.
- [Garcia, 1996] M. Garcia-Solaco, F. Saltor, M. Castellanos, "Semantic Heterogeneity in Multidatabase Systems", in *Object-oriented Multidatabase Systems*, O.A. Bukhres and A.K. Elmagarmid, ed-

- itors, Prentice Hall, 1996.
- [Garcia, 1997] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, V. Vassalos, J. Widom, "The TSIMMIS approach to mediation: Data models and Languages", *Journal of Intelligent Information Systems*, 1997.
- [Gardarin, 2002] G. Gardarin, A. Mensch, A. Tomasic, "An Introduction to the e-XML Data Integration Suite", in *Proceedings of EDBT 2002*, pp. 297-306, LNCS, 2002.
- [Geiger, 1995] K. Geiger, "Inside ODBC", Microsoft Programming Series, Microsoft Press, 1995.
- [Genesereth, 1997] M.R. Genesereth, A.M. Keller, O.M. Dushcka, "Infomaster: an information integration system", in *Proc. of ACM SIGMOD International Conference on Management of Data*, pp. 539-542, 1997.
- [Gligor, 1986] V. Gligor and R. Popescu-Zeletin, "Transaction Management in Distributed and Heterogeneous Database Management Systems", *Information System*, 11(4), pp. 287-297, 1986.
- [Gray, 1993] J. Gray, A. Reuter, "Transaction Processing: Concepts and Techniques", Morgan Kaufmann Publishing, 1993.
- [Hainaut, 1993b] J-L. Hainaut, M. Chandelon, C. Tonneau and M. Joris, Contribution to a Theory of Database Reverse Engineering, in *Proc. of the IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press, pp. 161-170, May 1993.
- [Hainaut, 1996] J-L. Hainaut, "Specification preservation in schema transformations - Application to semantics and statistics", *Data & Knowledge Engineering*, Elsevier Science Publish, 16(1), 1996.
- [Hainaut, 1996b] J-L. Hainaut, J. Henrard, J-M. Hick, D. Roland, V. Englebert, Database Design Recovery, in *Proc. of the 8th Conf. on Advanced Information Systems Engineering (CAiSE'96)*, Springer-Verlag, 1996.
- [Hainaut, 1999] J-L. Hainaut, Ph. Thiran, J-M. Hick, S. Bodart, A. Deflorenne, "Methodology and CASE tools for the development of federated databases", *International Journal of Cooperative Information Systems*, 8(2-3), pp. 169-194, World Scientific, June and September, 1999.
- [Hammer, 1997] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yernemi, M. Breunig, V. Vassalos, "Template-based Wrappers in the TSIMMIS System", in *SIGMOD Record*, 26(2), pp. 532-535, June 1997.
- [Härder, 1999] Th. Härder, G. Sauter, J. Thomas, "The Intrinsic Problems of Structural Heterogeneity and an Approach to their Solution", *The VLDB Journal*, Vol. 8, pp. 25-43, 1999.
- [Hasselbring, 1999] W. Hasselbring, "Top-down vs. Bottom-up engineering of federated information systems", in *Proceedings of EFIS'99*, pp. 131-138, Infix Verlag, 1999.
- [Heimbigner, 1985] D. Heimbigner, D. McLoed, "A Federated Architecture for Information System", *ACM Transactions on Office Information Systems*, 3(3), 1985.
- [Henrard, 1999] J. Henrard, J-L. Hainaut, J-M. Hick, D. Roland, V. Englebert, "Data structure extraction in database reverse engineering, in *REIS'99 Workshop*, 1999.
- [Hick, 2002] J-M. Hick, V. Englebert, J. Henrard, D. Roland, J-L. Hainaut, "The DB-MAIN Database Engineering CASE Tool (version 6.5) - Functions Overview", *DB-MAIN Technical manual*, Institut d'informatique, University of Namur, November 2002.
- [Hull, 1997] R. Hull, "Managing Semantic Heterogeneity in Databases: A Theoretical Perspective" in *Proc. of ACM PODS*, 1997.
- [Hurson, 1994] A. R. Hurso, M. W. Bright, H. Pakzad, "Multidatabase Systems: An Advanced Solution for Global Information Sharing", IEEE Computer Society Press, Los Alamitos, 1994.

- [Kaiser, 1992] G. Kaiser, C. Pu, "Dynamic Restructuring of Transactions", in *Transaction Models for Advanced Applications. Data Management Systems.*, A. Elmagarmid Ed., Morgan-Kaufman, 1992.
- [Keim, 1996] D.A. Keim, H-P. Kriegel and A. Miethsam, "Object-oriented querying of Existing Relational Databases", in *Fourth International Conference Database and Expert System Applications*, pp. 325-336, Springer-Verlag, 1993.
- [Larson, 1989] J. Larson, S.B. Navathe and R. El-Masi, "A Theory of Attribute Equivalence and its Applications to Schema Integration", *IEEE Transactions on Software Engineering*, 15(4), pp. 449-462, April 1989.
- [Lenzerini, 2001] M. Lenzerini, "*Data Integration is Harder than you Thought*", Slides of an invited talk in CoopIS'01, 2001.
- [Levy, 1995] A.Y. Levy, D. Srivastava, T. Kirk, "Data Model and Query Evaluation in Global Information Systems", *Journal of Intelligent Information Systems*, Special Issue on Networked Information Discovery and Retrieval, 5(2), pp. 121-143, 1995.
- [Levy, 1996] A. Levy, A. Rajamaran, J. Ordille, "Query Heterogeneous Information Sources Using Source Description", in *Proc. of the 22nd VLDB*, pp. 252-262, 1996.
- [Li, 2000] C. Li, E. Chang, "Query Planning with Limited Source Capabilities", in *Proc. of ICDE 2000*, pp. 401-412, 2000.
- [Liang, 1999] S. Liang, "*The Java Native Interface - Programmer's Guide and Specification*", The Java Series, Addison-Wesley, 1999.
- [Lim, 1998] E-P. Lim and R.H.L. Chiang, "A Global Object Model for Accomodating Instance Heterogeneity", in *Proceedings of ER'96*, LNCS, Vol. 1507, pp. 435-448, Springer-Verlag, 1998.
- [Lim, 1999] E-P. Lim, H-K. Lee, "Export Database Derivation in Object-oriented Wrappers", in *Information and Software Technology*, Vol. 41, pp. 183-196, Elsevier Science, 1999.
- [Litwin, 1986] W. Litwin, A. Abdellatif, "Multidatabase Interoperability", *IEEE Computer Magazine*, 19(12), pp.10-18, 1986.
- [Litwin, 1994] W. Litwin, "*Multidatabase Systems*", Prentice Hall: Englewood Cliffs, 1994.
- [Liu, 2000] D. Liu, K. Law, G. Wiederhold, "CHAOS: An Active Security Mediation System", in *Proceedings of CAiSE*, pp. 5 - 9, LNCS, Springer-Verlag, June 2000.
- [Maniola, 1998] F. Manola and al., "Supporting Cooperation in Enterprise-Scale Distributed Object Systems" in *Cooperative Information Systems - Trends and Directions*, M.P. Papazoglou and G. Schlageter editors, Academic Press, 1998.
- [Manolescu, 2001] I. Manolescu, D. Florescu, D. Kossman, "Pushing XML Queries inside Relational Databases", *Research Report*, INRIA Rocquencourt, 2001.
- [Markowitz, 1993] V.M. Markowitz, A. Shoshani, "Object queries over relational databases: Language, Implementation and Applications", in *Proceedings of the Ninth International Conference on Data Engineering*, IEEE Computer Sciences Press, 1993.
- [McBrien, 2000] P. McBrien, A. Poullovassilis, "Schema Evolution in Heterogeneous Database Architectures- A Schema Transformation Approach", in *Proceedings of CoopIS'00*, LNCS, Springer-Verlag, 2000.
- [Meng, 1995] W. Meng, C. Yu, "Query Processing in Multidatabase Systems", in W. Kim (editor) *Modern Database Systems*, Addison-Wesley, pp. 551-572, 1995.
- [Moss, 1985] J.E. Moss, "*Nested Transactions: an Approach to Reliable Distributed Computing*", The MIT Press, Cambridge, USA, 1985.

-
- [Mowbray, 1995] T. Mowbray and R. Zahavi, *"The Essential CORBA: Systems Integration Using Distributed Objects"*, Wiley, New-York, 1995.
- [Noël, 2001] B. Noël, *"Générateur de serveurs de Business Objects pour Wrappers"*, Mémoire de Graduat en Informatique, HEMES Liège, 2001.
- [Özsu, 1991] M.T. Özsu, P. Valduriez, *"Principles of Distributed Database Systems"*, Prentice Hall, New Jersey, 1991.
- [Özsu, 1999] M.T. Özsu, P. Valduriez, *"Principles of Distributed Database Systems "*, Prentice Hall, Second Edition, New Jersey, 1999.
- [Palopoli, 1999] L. Palopoli, G. Terracina, D. Ursino, "Semi-Automatic Techniques for Deriving Interscheme Properties from Database Schemes", *Data and Knowledge Engineering*, 30(3), pp. 239-273, 1999.
- [Parent, 1998] C. Parent and S. Spaccapietra, "Issues and Approaches of Database Integration", *Communications of the ACM*, 41(5), pp.166-178, 1998.
- [Parent, 2000] Ch. Parent and St. Spaccapietra, "Database Integration: the Key of Data Interoperability", in M.P. Papazoglou, S. Spaccapietra, Z.Tari, editors, *Advances in Object-Oriented Data Modeling*, MIT Press, 2000.
- [Papakonstantinou, 1995] Y. Papakonstantinou, A. Gupta, H. Garcia-Molina, J. Ullman, "A Query Translation Scheme for Rapid Implementation of Wrappers", *International Conference on Deductive and Object-Oriented Databases*, 1995.
- [Quian, 1995] X. Quian and L. Raschid, "Query Interoperation among Object-oriented and Relational Databases", in *Proceedings of the Eleventh International Conference on Data Engineering*, IEEE Computer Society Press, 1995.
- [Radulescu, 2001] A. Radulescu, C. Nicolescu, A.J.C. van Gemund, P. Jonker, "CPR: Mixed Task and Data Parallel Scheduling for Distributed Systems", in *Proceedings of IPDPS*, 2001.
- [Ram, 1999] S. Ram and V. Ramesh, "Schema Integration: Past, Present and Future. In A.K. Elmagarmid, A.Sheth and M. Rusinkiewicz, editors, *Management of Heterogeneous and Autonomous Database Systems*, pp. 119-155, Morgan Kaufmann, 1999.
- [Ramesh, 1995] V. Ramesh and S. Ram, "A methodology for interschema relationship identification in heterogeneous databases, *Proceedings of the Hawaii International Conference on Systems and Sciences*, pp. 263-272, 1995.
- [Ramesh, 1997] V. Ramesh and S. Ram, "Integrity Constraint Integration in Heterogeneous Databases: an Enhanced Methodology for Schema Integration, *Information Systems*, 22(8), pp. 423-446, 1997.
- [Reddy, 1998] P.K. Reddy and M. Kitsuregawa, "Reducing the Blocking in Two-Phase Commit Protocol Employing Backup Sites", in *Proceedings of Coopis '98*, IEEE Computer Sciences Press, 1998.
- [Reese, 1998] Reese, *"Database Programming with JDBC and JAVA"*, O'Reilly, Sebastopol, 1997.
- [Roantree, 2001] M. Roantree, J.B. Kennedy, P.J. Barclay, "Using a Metadata Software Layer in Information Systems Integration", in *Proceedings of CAiSE'01*, pp. 299-314, LCNS 2068, 2001.
- [Roland, 1997] D. Roland, J-L. Hainaut, "Database Engineering Process Modeling", in *Proc. of the Int Conference on The Many Facets of Process Engineering*, 1997.
- [Rosenthal, 1994] A. Rosenthal, D.S. Reiner, "Tools and Transformations - Rigorous and Otherwise - for Practical Database Design", *TODS*, 19(2), pp.167-211, 1994.
- [Roth, 1997] M. T. Roth, P. Schwarz, "Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources", in the *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

- [Rugaber, 1998] S. Rugaber and J. White, "Restoring a Legacy: Lessons learned", *IEEE Software*, 15(4):28-33, July-Aug 1998.
- [Rushby, 1983] J. Rushby, B. Randell, "A Distributed Secure System", *IEEE Computer*, 16(7):55-67, July, 1983.
- [Sandu, 1996] R.S. Sandu, E.J. Coyne, H.L. Feinstein and C.E. Youman, "Role-based access control models", *IEEE Computer*, 16(7):38-47, Feb. 1996.
- [Samaras, 1995] G. Samaras, K. Britton, A. Citron and C. Mohan, "Two-phase Commit Optimizations in a Commercial Distributed Environment", in *Journal of Distributed and Parallel Databases*, 3(4), 1995.
- [Sattler, 2002] K-U. Sattler, S. Conrad, G. Saake, "*Interactive Example-Driven Integration and Reconciliation for Accessing Database Federations*", Research Report, Magdeburg Universität, 2002.
- [Schmitt, 1996] I. Schmitt, G. Saake, "Integration of Inheritance Trees as Part of View Generation For Database Federations", in *Proceedings of ER'96*, pp. 195-210, 1996
- [Schwarz, 1999] K. Schwarz, I. Schmitt, C. Türker, M. Höding, E. Hildebrandt, S. Balko, S. Conrad, G. Saake, "Design Support for Database Federations", in *Proceedings of ER'99*, Paris, November 1999.
- [Schwarz, 1999b] K. Schwarz, I. Schmitt, C. Türker, M. Höding, E. Hildebrandt, S. Balko, S. Conrad, G. Saake, "Tool Support for the Design of Database Federations in SIGMA(FDB)", *Technical Report*, Magdeburg University, 1999.
- [Sheck, 1991] H-J. Sheck, G. Weikum, W. Schaad, "A Multi-level Transaction Approach to Federated DBMS Transaction Management" in Proc. 1st Workshop on Interoperability of Multidatabase Systems, pp. 280-287, *IEEE Computer Society Press*, 1991.
- [Sheth, 1989] A.P. Sheth and S. Gala, "Attribute relationships: an Impediment in Automating Schema Integration", In *Proceedings of the NSG Workshop on Heterogeneous Databases*, December 1989.
- [Sheth, 1990] A.P. Sheth and J.A. Larson "Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases", *ACM Computing Surveys*, 22(3):183-236, September 1990.
- [Sheth, 1991] A.P. Sheth, "Issues in Schema Integration: Perspective of an Industrial Researcher", *ARO Workshop on Heterogeneous Databases*, 1991.
- [Sheth, 1993] A. Sheth, S. Gala and S. Navathe, "On Automatic Reasoning for Schema Integration", *International Journal on Intelligent and Cooperative Information Systems*, 2(1), March 1993.
- [Sheth, 1993b] A. Sheth and V. Kashyap, "So far (schematically), yet so near (semantically)", in *Proceedings of the IFIP TC2/WG2.6 Conference on Semantics of Interoperable Database Systems, DS-5*, North-Holland, 1993.
- [Sheth, 1990] A.P. Sheth and J.A. Larson "Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases", *ACM Computing Surveys*, 22(3):183-236, September 1990.
- [Sneed, 1997] H.M. Sneed, "Program Interface Reengineering for Wrapping", in Proc. of the 4rd IEEE Working Conf. on Reverse Engineering, IEEE Computer Society Press, 1997.
- [Souder, 2000] T. Souder and S. Mancoridis, "A Tool for Securely Integrating Legacy Systems into a Distributed Environment", in *Proceedings of WCRE'00*, IEEE Computer Society Press, 2000.
- [Spaccapietra, 1991] S. Spaccapietra and C. Parent, "Conflicts and correspondence assertions in interoperable databases", *SIGMOD Record*, 20(4), pp. 49-54, December 1991.
- [Spaccapietra, 1994] S. Spaccapietra and C. Parent, "View Integration - A Step Forward in Solving

- Structural Conflicts", *IEEE Transactions on Knowledge and Data Engineering*, 6(2), pp. 258-274, 1994.
- [Templeton, 1995] M. Templeton, H. Henley, E. Maros, D.J. Van Buer, "InterVisio: Dealing with the Complexity of Federated Database Access", *The VLDB Journal*, 4(2), pp. 287-317, 1995.
- [Thiran, 1998] Ph. Thiran, J-L. Hainaut, S. Bodart, A. Deflorenne, J-M. Hick, "Interoperation of Independent, Heterogeneous and Distributed Databases. Methodology and CASE Support: the InterDB Approach" in *Proceedings of CoopIS'98*, IEEE, New-York, August 1998.
- [Thiran, 1999] Ph. Thiran, J-L. Hainaut, J-M. Hick, A. Chougrani, "Generation of Conceptual Wrappers for Legacy Databases", in *Proceedings of DEXA'99*, LCNS, Springer-Verlag, September 1999.
- [Thiran, 2001a] Ph. Thiran, J-L. Hainaut, "Interoperability of Legacy Databases - A Combined Top-Down and Bottom-Up Approach", in *Proceedings of the Doctoral Consortium of CAiSE '01*, 2001
- [Thiran, 2001b] Ph. Thiran, J-L. Hainaut, "Wrapper Development for Legacy Data Reuse", in *Proceedings of WCRE '01*, 2001
- [Thiran, 2001c] Ph. Thiran, J-L. Hainaut, "Evolving Hybrid Databases: Architecture and Methodology", in *Proceedings of EFIS '01*, October 2001.
- [Terracina, 2000] G. Terracina and D. Ursino, "Deriving Synonymies and Homonymies of Object classes in semi-structured Information Sources", in *Proc. of International Conference on Management of Data (COMAD 2000)*, pp. 21-32, McGraw Hill, 2000.
- [Tomasic, 1996] A. Tomasic, L. Raschid, P. Valduriez, "Scaling Heterogeneous Databases and the Design of Disco", in *Proceedings of the International Conference on Distributed Computer Systems*, 1996.
- [Türker, 1999] C. Türker, "*Semantic Integrity Constraints in Federated Database Schemata*", PhD Thesis, Magdeburg Universität, Infix Press, 1999.
- [Ullman, 1997] J.D. Ullman, "Information Integration Using Logical View", in *Proc. of ICDE '97*, volume 1186 of LNCS, pp. 19-40, Springer-Verlag, 1997.
- [Umar, 1997] A. Umar, "*Application (Re)Engineering - Building Web-Based Applications and Dealing with Legacies*", Prentice Hall, 1997.
- [Urban, 1991] S. Urban, J. Wu, "Resolving Semantic Heterogeneity Through the Explicit Representation of Data Model Semantics", *SOGMOD Record*, 20(4), pp. 55-58, 1991.
- [van den Heuvel, 2000] W.J. van den Heuvel, W. Hasselbring, M. Papazoglou, "Top-Down Enterprise Application Integration with Reference Models" in *Proceedings of EFIS'00*, pp. 11-22, IOS Press and Infix, 2000.
- [Vermeer, 1996] M.W.W. Vermeer and P.M.G. Apers, "On the Applicability of Schema Integration Techniques to Database Interoperation", in *Proc. Of 15th Int. Conf. On Conceptual Modeling*, ER'96, Cottbus, pp. 179-194, Oct. 1996.
- [Vermeer, 1997] M.W.W. Vermer, "*Semantic Interoperability for Legacy Databases*", PhD Thesis, Twente University, 1997.
- [Vidal, 1998] M.E. Vidal, L. Raschid, J-R. Gruser, "A Meta-Wrapper for Scaling up to Multiple Autonomous Distributed Information Sources", in *Proc. of CoopIS'98*, pp. 148-157, IEEE Computer Sciences Press, 1998.
- [Wiederhold, 1992] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, pp. 38-49, March 1992.
- [Wiederhold, 1995] G. Wiederhold, "Value-Added Mediation in Large-Scale Information Systems",

IFIP Data Semantics (DS-6), Atlanta, Georgia, 1995.

[Wiggerts, 1997] T. Wiggerts, H. Bosma, E. Felt, "Scenarios for the Identification of Objects in Legacy Systems", in *Proceedings of WCRE '97*, IEEE Computer Society Press, 1997.