FACULTES UNIVERSITAIRES
NOTRE-DAME DE LA PAIX NAMUR

Institut d'Informatique

Legacy Database Federation
A Combined Forward-Reverse Approach

Philippe Thiran

University of Namur - Institut d'Informatique
rue Grandgagnage, 21 • B-5000 Namur (Belgium)

Jury: Prof. Jean Fichefet - University of Namur (President)
Prof. Christine Parent - EPFL, University of Lausanne, Switzerland
Prof. Amit Sheth - University of Georgia, Athens, USA
Prof. Vincent Englebert - University of Namur
Prof. Jean-Luc Hainaut - University of Namur (PhD supervisor)

October, 2003

Thesis presented in order to obtain a PhD degree in Science, Computer Science Option

A mon Hélène

L'Amour a toujours été pour moi
la plus grande des affaires, ou plutôt la seule
Stendhal

Preface

I would like to express my appreciation to many colleagues and friends who contributed to this thesis in any way.

First, I would like to thank my thesis advisor Prof. Jean-Luc Hainaut for giving me the opportunity to carry out the research described in this thesis. He motivated me to work on my PhD thesis and on numerous scientific reports and papers reporting on the intermediate research results. Working with him has always been a very instructive as well as a pleasant experience for me. During the week and even weekends, and in spite of his overbooked agenda, he was always willing to supervise my work and give me advice. Without his supervision, I would never have been able to achieve this result.

I would like also to express my sincere gratitude for the involvement of Prof. Vincent Englebert in my research. He is thanked for the stimulating discussions and for reviewing the early draft version of my work.

Prof. Christine Parent and Prof. Amit Sheth also deserve a special gratitude for reviewing my thesis and giving some advice on improving the final version.

I wish to thank everyone in the institute of Informatics of the university of Namur for their support of my work and providing a pleasant and motivating scientific research atmosphere. Especially, I gratefully acknowledge the contributions of the current and former members of the DB-MAIN research team. In addition, I would like to thank the graduate students for their help during the implementation and test of the prototypes: Bernard Noël and Denis Renaud deserve special thanks.

Other people have also contributed to the research put forward in this thesis. Discussions with the following researchers and practitioners in particular contributed to my understanding of this research: Prof. Djamal Benslimane, Prof. Gunter Saake, Prof. Stefano Spaccapietra, dr. Willem-Jan van den Heuvel.

I wish to thank the *Région Wallonne* and more in particular *la DGTRE*, for partially funding this research project.

I would like to thank my little family for their ongoing support and love. My wife and my baby always supported me. They helped me in many ways, especially during the stressful time of completing this work. Without their support throughout the years, I would never have been able to deliver this thesis. You both are sources of inspiration which make life so worth living.

Eindhoven
September 2003

Table of Contents

Detailed Table of Contents

Chapter 1

Introduction

In which the reader is introduced to mediation of legacy databases by first giving its main issues. The terms legacy, autonomy, heterogeneity, mediation and federation are defined. The current solutions are then presented and studied. An overview of the approach developed in this thesis 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 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*.

We refer to software services allowing such so-called legacy database systems to cooperate, as providing *interoperability.* Such services provide users and application programs with an integrated view of data dispersed over various component databases.

In this thesis, we focus on the interoperability of *legacy and heterogeneous databases*. We introduce the thesis by first giving the main issues about interoperability. Next, a short overview of the interoperability research is presented. We then develop a small example that il-

lustrates some of the problems we intend to address. Finally, we present the purpose and the topic of the thesis.

1.2 Problem and Context

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 Information System (IS) is any IS that significantly resists modifications and change. 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 corporations and are thus very valuable from a business point of view. Legacy data systems contain very valuable information that is embedded in legacy databases/flat files and application codes [Umar, 1997]. In many cases, legacy data systems are the only source of years of business rules, historical data, and other valuable knowledge. Access to this information is of vital importance to new and emerging tools and applications [Bouguettaya,1998]. 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 that encapsulates the underlying data and mediates between the legacy system and the new program interface [Rugaber, 1998].

Dealing with such systems is very costly because of the complexity of understanding data semantics which is either buried in application programs or was 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 unmormalized 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 can 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 the 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 can 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 keep 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 [Özsu, 1991]. *Location independence* isolates the user from knowing where the data reside. 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. Finally, *semantic level independence* solves the problem of multiple, replicated and conflicting representations of similar facts. Despite much effort spent by the scientific community, *semantic independence* still is an open and largely unsolved problem ([Aslan, 1999], [Härder, 1999], [Sattler, 2003]).

Semantic heterogeneity

Current legacy databases can model overlapping universes of discourse. The same real-world objects can be represented in the multiple legacy databases. Identifying similar or even same elements is a non-trivial task due to several reasons, for instance, conflict problems such as *homonyms*, which occur when the same name is used for different real-world concepts, and *synonyms*, which occur when different names are used for the same real-world concept. Since semantics is *relative* [Garcia, 1996], two legacy databases that are intended to model the same part of the real world can exhibit differences in the way the real world is represented. In this context, the term semantic heterogeneity is principally caused by the design autonomy of the legacy databases:

- *Database conceptualization:* each database designer has its own conceptualization of the universe of discourse. The perspective and focus of the database designer determine the result of the conceptualization. Different concepts can be used to express the same real-world object or relationship. Legacy databases are usually designed by different database designers at different times. Therefore, the existence of the semantic heterogeneity is the usual cause.
- *Database schema design:* semantic heterogeneity can be the consequence of the use of different data models for representing the same real-world concepts in the database. If the different database models provide structures that cannot be mapped into equivalent structures of the other data model, arbitrary representations cannot be transformed to each other without loss of information. Database model also allows alternative ways to describe the same universe of discourse. Therefore, semantic heterogeneity can even occur if the same data model is used for modeling the same universe of discourse.

Leveling the semantic heterogeneity among databases thus implies to capture beforehand their semantics. Their acquisition is all the more important as the databases are legacy.

1.2.5 Mediation

To address the problem of interoperability of information systems in general, the term *mediation* has been defined [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], [Meng, 1995], [Vermeer, 1996], [Genesereth, 1997], [Cluet, 1998]).

1.2.6 Legacy Database Federation

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 these databases.

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

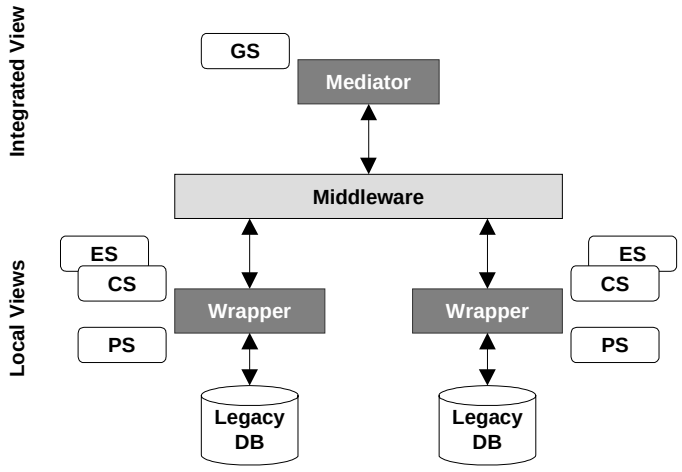


Figure 1-1: A general architecture of database federation. The architecture is made up of a hierarchy of wrapper components (wrapper and mediator) and a hierarchy of schemas (physical schema - PS; conceptual schema - CS; export schema - ES and global schema GS).

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* (PS) from which a semantically rich description called *conceptual schema* (CS), is obtained through a database reverse-engineering process. From this conceptual view, a subset called *export schema* (ES) is extracted. All the export schemas are merged into the *global (or fed-*

erated) schema (GS). The global schema as well as the conceptual and export schemas are expressed in a *canonical data model* 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 detail about the location and representation of relevant data to applications.

A *wrapper* tops each legacy database. Its aim is to hide the data model and the query language heterogeneity to mediators. 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 Federation and Methodologies

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] and [Batini, 1986], the database integration is made up of five main processes: (1) pre-integration; (2) schema preparation; (3) schema comparison; (4) schema conformation and (5) schema merging:

- *Pre-integration.* In this step, the schemas of the participating databases are analyzed and the integration strategy is decided. In general, a distinction is made between binary and n-ary integration strategies. Binary integration strategies allow the simultaneous integration of exactly two schemas whereas in the n-ary case more than two schemas can be integrated in one step. Furthermore, one-shot and iterative strategies are distinguished.
- *Schema preparation.* In order to compare schemas, they have to be translated into a common data model. This activity is know as model transformation ([McBrien, 1999], [Hainaut, 1996]). Ideally, model transformation preserves the semantics of the schemas.

- *Schema comparison.* The schemas are then analyzed and compared to identify similar components. Often, the similarities are stated in form of the correspondence assertions [Spaccapietra, 1992]. Moreover, structural and semantic conflicts such as type, naming or scaling discrepancies are detected. Schema comparison is primarily performed by human beings. These persons are assumed to be experts in the application domain and thus have a certain kind of inter-schema knowledge. A complete automation of this process is impossible [Navathe, 1996].
- *Schema conformation.* The objective of schema conformation is to make the source schemas compatible for integration. For that, schematic and semantic conflicts detected in the preceding step have to be solved in this step. Solving the conflicts occurring in heterogeneous databases has been studied in numerous references, by e.g. [Spaccapietra, 1991], [Batini, 1986], [Vermeer, 1996], [Kashyap, 1997]. 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]. Often, a general automatic conflict resolution is not feasible; interactions with the analyst are needed. In order to resolve a conflict, the analyst must understand the semantic relationships among the concepts involved in the conflict. Ideally, schema conformation produces a set of schemas without any schema conflicts.
- *Schema merging.* Once the source schemas have been compared and conformed, schemas are merging into one global schema.

The database integration approach produces the structure of the global schema that depends directly on the integrated export schemas and on the integration method used [Busse, 2000]. As discussed in [Hasselbring, 1999], this bottom-up approach exhibits the following problems:

- The process of integration is often more complex than required for the actual requirements of organizations. Since the relevant information is hidden in a global (or federated) schema, the user is responsible for finding the required information.
- It rarely considers the requirements from the new applications that are to be developed on top of the legacy databases.
- It is not suitable for frequent dynamic changes of organization requirements since the global (or federated) schema is static or usually too difficult to change.

1.2.8 Database Federation and Mappings

One of the most challenging issues in federated databases is the definition of the mappings. Two main basic approaches have been used to specify them. The first and very widespread approach ([Vermeer, 1996], [Garcia, 1997]) is *query-oriented* in that it provides mechanisms by which users define global schema constructs as view over source schema constructs, but does not focus on the semantics of the data sources. More recent approaches on automatic

wrapper generation ([Vidal, 1998], [Hammer, 1997]) also are query-oriented.

In contrast, the second approach ([Thiran, 1998], [McBrien, 1998]) is *schema-oriented* in that mappings are defined as schema transformations that are used to automate the translation of queries. A comparison of these approaches is reported in [McBrien, 2002]. The schema-oriented approach has the further advantage of decomposing the transformation of schemas into a sequence of small steps, whereas the query-oriented approaches required that constructs in one schema are directly defined in terms of those in the other schema.

1.2.9 Database Federation and Wrappers

Current database federation architectures ([Vermeer, 1996], [Hammer, 1997], [Bouguettaya, 1998], [McBrien, 1999]) consider a wrapper as a model converter, i.e., a software component that translates data and queries from one data model, generally the legacy DMS model, to another, abstract, DMS-independent, model. That is, the wrapper is only used to overcome the data model heterogeneity in a database federation.

Such a wrapper is based on the quality and completeness of the database structures to be wrapped that cannot be relied on when dealing with legacy databases [Thiran, 2001]. The wrapper should offer a semantically richer description of the underlying database than that provided by the DDL statements. For instance, a wrapper that interfaces a COBOL file collection should ensure referential integrity implied by implicit (undeclared) foreign keys that exist between the record types.

1.3 Scope of the thesis

We mainly focus on the *legacy* and *semantic* aspects of database federations. Our primary goal of this thesis is to analyze these aspects in detail and to present architectural and methodological solutions to these aspects. This is necessary in order to be able to better understand and automate the federation database development.

We do not assume the global schema as the result of a bottom-up process. Referring to [van den Heuvel, 2000] and [Busse, 2000], we state that the global schema is defined not only by the contribution of the legacy databases but also includes the new requirements. Our goal is to resolve the database integration by combining *forward and reverse processes*. We therefore develop an integration methodology based on a conceptual data description and intended to find out which part of the actual requirements can be covered by the legacy systems and which part has to be managed by additional systems.

Motivated by the legacy aspect of the database federation, we investigate the close link between the *database reverse engineering process* and the *wrapper development.* Our goal is to propose a wrapper architecture that provides a semantically rich description of the underlying database. Moreover, since we take into account the actual requirements of an organization,

we also study the integration of these requirements into the wrapper specification.

One of the most challenging issues is the *definition of the mappings* between all the schemas of a federation. In this thesis, we develop a formal transformational approach that is built on an unique extended entity object-oriented model from which several abstract submodels can be derived by specialization. This approach is intended to provide an elegant way to unify the multiple models and mapping descriptions of the federation.

In the following, we briefly summarize the basic assumptions and restrictions underlying this thesis:

- First of all, since we are mainly interested by the legacy aspects of database federation, we concentrate on the wrapper architecture. We do not therefore address the architectural issues of mediators. Fur such issues, we refer to [Meng, 1995], [Genesereth, 1997] or [Cluet, 1998].
- Moreover, we do not address issues such as infrastructure and transaction. For such issues, we refer to [Sheck, 1991], [Deacon, 1996] and [Özsu, 1999].
- Finally, we concentrate on the structural part of the legacy schemas to be integrated. Dynamic constraints which describe restrictions on the legacy database behavior are beyond the scope of this thesis. For an overview and discussion on behavior integration, we refer to [Thieme, 1995] and [Vermeer, 1996].

1.4 Motivations

In this section, we develop a small example that illustrates some of the problems we intend to address in this thesis. We consider a company in which two manufacturing sites M1 and M2 are active. We also consider the personnel departments P1 and P2 that ensure the HRM of each of these sites, and the sales department S, common to both. Due to historical reasons, the personnel and sales functions of the company are controlled by three independent databases, namely DB-P1 (personnel of site M1), DB-P2 (personnel of site M2) and DB-S (sales of sites M1 and M2). Though the databases are independent, the management applications involve data exchange through asynchronous text files transfer. From a technical point of view, database DB-P1 is made up of a collection of standard COBOL files, while DB-P2 was developed in Oracle V5[1]. DB-S was recently (re)developed with a modern version of IBM DB2.

The new organizational trends force the company to reconsider the structure and objectives of its information system. First, additional functions must be developed to meet new requirements, notably in customer management. Secondly, the existing functions must be integrated, so that the supporting databases are required to be integrated too.

The scenario, according to which a quite new system encompassing all the functions of per-

1. This version of Oracle ignored the concepts of primary and foreign keys.

sonnel, sales and customer management is built, must be discarded due to too high organizational and financial costs. In particular, the legacy databases cannot be replaced by a unique system, nor even can be reengineered. The company decides to build a virtual database comprising (1) excerpts from the databases DB-P1, DB-P2, DB-S and (2) a new database DB-C that is to support the customer management department, and that will be developed with the object-relational technology. This new architecture will allow new applications to be developed against a unique, consistent, integrated database. It is also decided that some local legacy applications are preserved. This objective raises several critical problems on the distribution of the general requirements and the responsibilities of the whole system among the legacy and new components. Another problem also appears: how to bring together the legacy databases and the new requirements?

It is decided to address one integration problem at a time as follows (Figure 1-2).

- First, each personnel database is provided with a specific wrapper that yields a semantically rich abstract view of its contents according to a common model (Wrapper P1, Wrapper P2). In particular, these wrappers make explicit, and manage, hidden constructs and constraints such as foreign keys, that are unknown in the COBOL and Oracle V5 models. Similarly, a wrapper is developed for the DB-S database according to this abstract model (Wrapper S). The main problem in building these wrappers is to recover the semantics of the legacy databases expressed in their conceptual schemas (LCS-P1, LCS-P2, LCS-S) from their physical schemas (LPS-P1, LPS-P2, LPS-S) through reverse engineering techniques. It must be noted that these wrappers export legacy data structures according to the needs of the future integrated system.
- Then, a common mediator is built on top of these wrappers to reconcile both personnel databases. This component is in charge of integrating the data from both databases by solving data format conflicts, semantic conflicts and data duplication conflicts (the employees that work on both sites are represented in both databases). This unique personnel database is known through its federated conceptual schema FCS-P.
- All the databases of the current system are unified under a common mediator that manages the semantic links between the (abstract) personnel database and the sales database. This component completes the structure of the federated database built on the three legacy databases DB-P1, DB-P2 and DB-S. A common federated global conceptual schema is available, namely FGS-PS.
- By comparing the services supplied by the federated database against the requirements of the whole system the company wants to develop, and expressed through its global conceptual schema NGS-PSC, the minimum requirements of the new components are elicited. From the corresponding conceptual schema NCS-C, a new object-relational database DB-C is developed, with physical schema NPS-C.
- Finally, in order to provide new applications with a unique database, a global mediator is built, that integrates the federated and the new databases, and that offers a straightforward materialization of the conceptual schema NGS-PSC. Whether the new database is accessed through a wrapper or not, depends on the distance between its data model and

the abstract model provided by the mediators. In this example, the Mediator PS model and the DBMS model both are Object-relational. Therefore, the new database need not be wrapped.

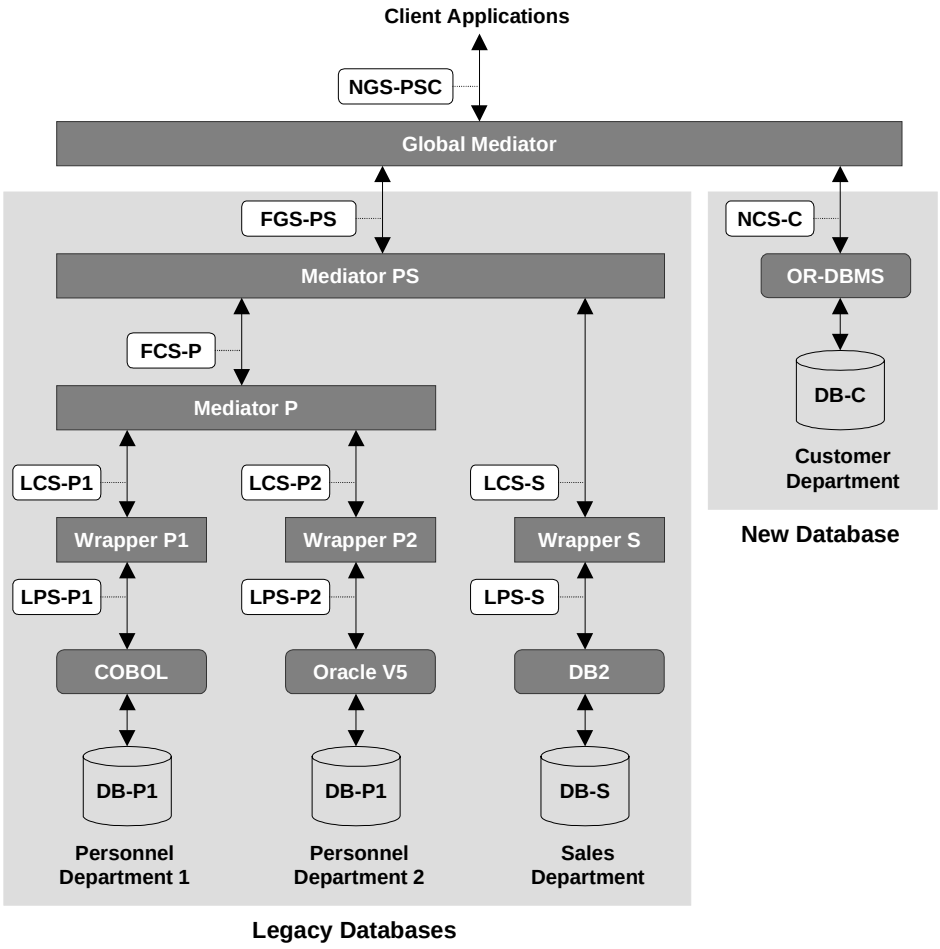


Figure 1-2: The new system, available as a unique integrated database with schema NGS-PSC, will comprise a federated database that abstracts and integrates three independent legacy database (COBOL files DB-PP1, Oracle 5 DB-P2, DB2 DB-S), and the new object-relational database DB-C.

1.5 Overview

Thesis topic

We can now define the topic of this thesis as follows:

> *How can we provide semantic mediation for legacy databases and new databases to be developed using a combined forward and reverse methodology?*

We distinguish four main tasks addressing this question: (1) defining a generic framework intended to express all the federation schemas and mappings; (2) defining a federation architecture; (3) developing a methodology based on a forward/reverse approach; and (4) proposing CASE tools that support both modeling activities and architecture component building.

Generic integration framework

This issue is discussed in Part I of this thesis. It involves defining a unique and generic framework intended to express all the schemas and mappings of a federation. This part is based on the work first presented in [Hainaut, 1996] and [Hainaut, 1999].

In *Chapter 2*, 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, wrapper schemas as well as global 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 3*, we define the mappings as *schema transformations*. We present the concepts and properties of schema transformations. An inventory of useful transformations is presented. These transformations are used to automate the translation queries between the schema hierarchy. We finally introduce the notion of schema transformation history.

Federation architecture

This issue is discussed in Part II of this thesis. We assume a federation architecture (wrapper/mediator) combined with new components architecture (mediator and new DMS). We concentrate on the legacy aspects of the databases to be integrated. This part is based on the work first presented in [Thiran, 1998], [Thiran, 2001b] and [Thiran, 2003].

In *Chapter 4*, we present a schema-oriented framework for database federation that allows mediation tasks and responsibilities to be split up and distributed. We discuss the important role of the *wrapper* in the particular case of a federated database architecture that integrates new requirements. In this thesis, we argue that some responsibilities commonly allocated to

mediators can be transferred to wrappers. To this end, we propose a wrapper/mediator architecture where the wrapper is more than a model and query converter.

In *Chapter 5*, we present and develop the technology of *wrappers for legacy databases*. We discuss how queries against a wrapper can be translated into queries against its underlying legacy database. Our experience in building wrappers is then presented and some baselines in wrapper generation are discussed. The architecture of operational wrappers for relational and standard file databases - the InterDB wrappers - is finally presented.

Forward-reverse methodology

This issue is discussed in Part III of the thesis. We consider that the global schema is defined not only by the contribution of the legacy databases but also includes the new requirements. It is the answer to the requirements of the organization as they are perceived from now on. This part is based on the work first presented in [Hainaut, 1999] and [Thiran, 2001c].

In *Chapter 6*, we propose to resolve the integration by combining forward and reverse processes. Unlike [van den Heuvel, 2002], we believe that reverse and forward processes are tightly bound. Since the approach is based on reusing existing resources as far as possible, the future system will comprise legacy databases as well as a new one, so that the requirements will be met by both kinds of databases. Therefore, one of the challenging problems is the precise distribution of the requirements among these components.

CASE support

This issue is discussed in Part IV of the thesis. Like any complex process, building a database federation cannot be successful without the support of adequate tools called CASE tools. Nevertheless, completely automating the process is unrealistic for real world systems. Hence, the need for computer-based assistance tools which address several aspects of the federation development. This part is based on the work first presented in [Hainaut, 1999] and [Thiran, 2000].

In *Chapter 7*, we present the main requirements that CASE tools should meet for the development of database federation systems, and presents an operational CASE tool (DB-MAIN) and its extensions which are intended to address some of the requirements.

Finally, *Chapter 8* presents our conclusions.

Part I

Generic Integration Framework

Chapter 2

Generic Data Model

In which the generic data model of federation schemas 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.

2.1 Introduction

Over the years, several data models have been used to build legacy databases: simple files, hierarchical, network, relational and object-oriented models. To overcome the data model heterogeneity among legacy databases, schemas that correspond to their models are translated into schemas using a *Canonical Data Model* (CDM). That allows for resolving syntactic heterogeneity. For example, in the Multibase system [Dayal, 1982b], the legacy DMS are relational and network systems, while the CDM follows the functional model.

It is usually expected that the modeling power of the CDM is as expressive as that of the legacy data models. The *relational model* has frequently been used as the CDM for relational, hierarchical and network databases ([Türker, 1999], [Rosenthal, 1985]). 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 [Cardenas, 1987]. 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], [Keim, 1996], [Vermeer, 1996], [Roantree, 2001]). The shift has been motivated by the fact that the object-oriented model can be used as a tool for both design and implementation. The recent trend 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 information 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 this thesis, we define a high-level generic data model, namely the *generic data model*, such that it is possible to represent the schemas 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. As a result, the generic data model is the ideal support for schema transformations (Chapter 3). 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 generic data model has been defined in [Hainaut, 1989] and implemented in the DB-MAIN repository (Chapter 7). Its most important components are presented in this chapter.

2.2 Main Concepts

The main concepts of the generic data model are described in the following sections. Figure 2-1 summarizes the seven major constructs.

Schema	Collection of data or information structures
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 2-1: The seven main concepts of the generic data model.

2.2.1 Schema

A schema is a description of a collection of data or information structures. It mainly comprises entity types (or object types), relationship types, attributes, domains, collections, processing units and various constraints (expressed as properties of groups of components).

2.2.2 Entity Type

An entity type represents a class of concrete or abstract real-world entities, such as customers, orders, books. It can also be used to model more computer-oriented constructs such as record types, tables, segments, and the like. This interpretation depends on the abstraction level of the schema, and on the data model.

Figure 2-2 shows an example of an entity type named Customer. The top compartment contains the name. The second compartment contains some attributes that characterize the entity type. The third compartment contains various constraints, each holding among the components of a group. The bottom compartment contains some processing units applicable to the entity type. Only the first compartment is mandatory, while the others are independently optional. These attributes, constraints and processing units are examined hereafter.

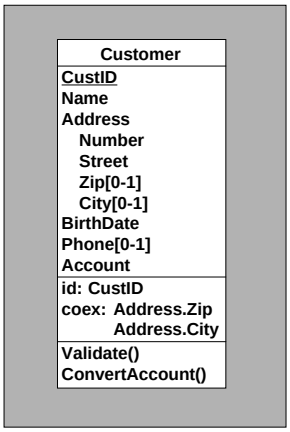


Figure 2-2: An example of an entity type.

In an object-oriented model, we will use the term *object class* instead. Object classes generally are given methods and appear in ISA hierarchies.

An entity type can be a subtype of one or several other entity types, called its *super-types*. If F is a subtype of E, then each F entity is an E entity as well. The collection of the subtypes of an entity type E is declared total (symbol T) if each E entity belongs to at least one subtype; otherwise, it is said to be partial. This collection is declared disjoint (symbol D) if an entity of a subtype cannot belong to another subtype of E; otherwise, it is said to overlap. If this collection is both total and disjoint, it forms a partition (symbol P)

An entity type can comprise attributes, can play roles in rel-types, can be collected into collections, can be given constraints (through groups).

Since a supertype/subtype relation is interpreted as *"each F entity is an E entity"*, it is called an ISA relation. ISA relations form what is called an ISA *hierarchy*.

The four supertype/subtype patterns can be summarized in Figure 2-3, where B1 and B2 are two subtypes of A.

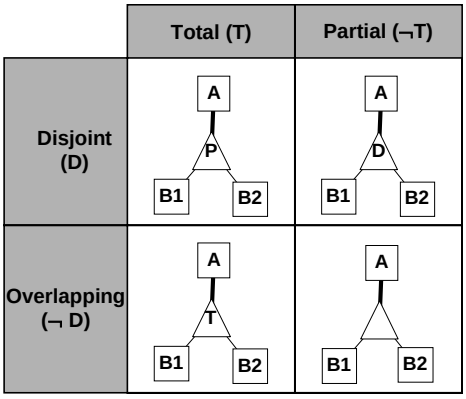


Figure 2-3: The four patterns of ISA hierarchy.

2.2.3 Relationship Type

A *relationship type* represents a class of associations between entities. It consists of entity types, each playing a specific *role*. A rel-type with 2 roles is called *binary*, while a rel-type with N > 2 roles is generally called *N-ary*. A rel-type with at least 2 roles taken by the same entity type is called *cyclic*.

Normally, a role is played by one entity type only. However, a role can be taken by more than one entity type. In this case, it is called a *multi-ET* role.

Each role is characterized by its *cardinality* [i-j], a constraint stating that any entity of this type must appear, in this role, in i to j associations or relationships. Generally i is 0 or 1, while j is 1 or N (= *many* or *infinity*). However, any pair of integers can be used, provided that $i \leq j$, $i \geq 0$ and $j > 0$.

A binary rel-type R between A and B with cardinality [i1-j1] for A, [i2-j2] for B (Figure 2-4) is called:

- one-to-one if j1 = j2 = 1
- one-to-many from A to B if j1 > 1 and j2 = 1
- many-to-one from A to B if j1 = 1 and j2 > 1
- many-to-many if j1 > 1 and j2 > 1
- optional for A if i1 = 0
- mandatory for A if i1 > 0.

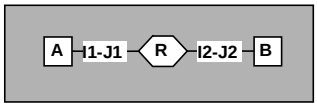


Figure 2-4: A relationship type between two entity types.

A rel-type can have attributes, and can be given constraints (through groups) and processing units.

2.2.4 Attribute

An attribute represents a common property of all the entities (or relationships) of a given type. Each attribute is characterized by its *cardinality* [i-j], a constraint stating that each parent has from i to j values of this attribute. Generally i is 0 or 1, while j is from 1 to N (= infinity). However, any pair of integers can be used, provided $i \leq j$, $i \geq 0$ and $j > 0$. The default cardinality is [1-1], and is not represented graphically. An attribute with cardinality [i-j] is called:

- single-valued if j = 1
- multivalued if j > 1
- optional if i = 0
- mandatory if i > 0

Simple attribute

Simple attributes have a *value domain* defined by a data type (number, character, boolean, date,...) and a length (1, 2,..., 200,..., N [standing for *infinity*]). These attributes are called *atomic*. For example in Figure 2-2, CustID, Name, Address, BirthDate, Phone and Account are attributes of an entity type Customer.

If the value domain has some specific characteristics, it can be defined explicitly as a *user-defined* domain, and can be associated with several attributes of the project. A user-defined domain is atomic or compound.

Compound attribute

An attribute can also consist of other component attributes, in which case it is called *compound*. The *parent* of an attribute is the entity type, the relationship type or the compound attribute to which it is directly attached. An attribute whose parent is an entity type or a rel-type is said to be at level 1. The components of a level-i attribute are said to be at level i+1. Number, Street, Zip and City are all sub-attributes of the attribute Address. Being made up of meaningful components, Address is a compound attribute.

Multivalued attribute

A plain multivalued attribute represents sets of values, i.e., unstructured collections of distinct values. In fact, there exists six categories of collections of values:

- *Set:* unstructured collection of distinct elements (default).
- *Bag:* unstructured collection of (not necessarily distinct) elements.
- *Unique list:* sequenced collection of distinct elements.
- *List:* sequenced collection of (not necessarily distinct) elements.
- *Unique array:* indexed sequence of cells that can each contain an element. The elements are distinct.
- *Array:* indexed sequence of cells that can each contain an element.

These categories can be classified according to two dimensions: uniqueness and structure.

	Unstructured	**Sequence**	**Array**
Unique	(set)	ulist	uarray
Not unique	bag	list	array

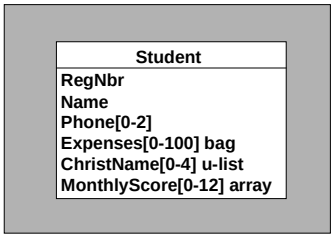


Figure 2-5: Some non-set multivalued attributes. While Phone defines a pure set, Expenses represents a bag, Christ(ian)Name a list of distinct values and MonthlyScore an array of 12 cells, of which from 0 to 12 can be filled.

2.2.5 Group

A group is made up of components, which are attributes, roles and/or other groups. A group represents a construct attached to a parent construct, i.e., to an entity type, a rel-type or to a multivalued compound attribute. It is used to represent concepts such as identifiers, foreign keys, indexes, sets of exclusive or coexistent attributes. A group of an entity type can comprise inherited attributes and roles, i.e., components from its direct or indirect supertypes.

It can be assigned one or several *functions* among the following:

- *Primary identifier:* the components of the group make up the *main identifier* of the parent construct; it appears with symbol id. An parent construct has at most one primary identifier. It is made up of mandatory attributes and/or roles. In Figure 2-6, each entity type has a primary identifier group either composed of a simple attribute (BookID of the

entity type Book) or made up of an attribute and a role (of.Book and SerialNbr of the entity type Copy).

- *Secondary identifier:* the components of the group make up a *secondary identifier* of the parent construct; it appears with symbol id'; a parent construct can have any number of secondary id. In Figure 2-6, the entity type Book has a secondary identifier made up of two attributes (Title and Publisher).
- *Coexistence:* the components of the group must be *simultaneously present or absent* for any instance of the parent construct; the group appears with symbol coex; all its components are optional. For instance, in Figure 2-6, the entity type Copy contains a coexistence group made up of two optional attributes State and StateComment. It states that both attributes are valued or void
- *Exclusive:* among the components of the group *at most one must be present* for any instance of the parent construct; the group appears with symbol excl; all its components are optional.
- *At-least-1:* among the components of the group, *at least one must be present* for any instance of the parent construct; the group appears with symbol at-lst-1; all its components are optional.
- *Exactly-1:* among the components of the group, *one and only one must be present* for any instance of the parent construct (= exclusive + at-least-1); the group appears with symbol exact-1; all its components are optional.
- *Access key:* the components of the group form an *access mechanism* to the instances of the parent construct (generally an entity type, to be interpreted as a table, a record type or a segment type); the access key is an abstraction of such constructs as indexes, hash organization, B-trees, access paths, and the like; it appears with symbol acc or access key. In Figure 2-6, several groups of the entity type Copy are tagged acc.
- *User-defined constraint:* any function that does not appear in this list can be defined by the user by giving it a name; some examples: at-most-2 (no more than two components can be valued), lhs-fd (left-hand-side of a functional dependency), less-than (the value of the first component must be less than that of the second one), etc.

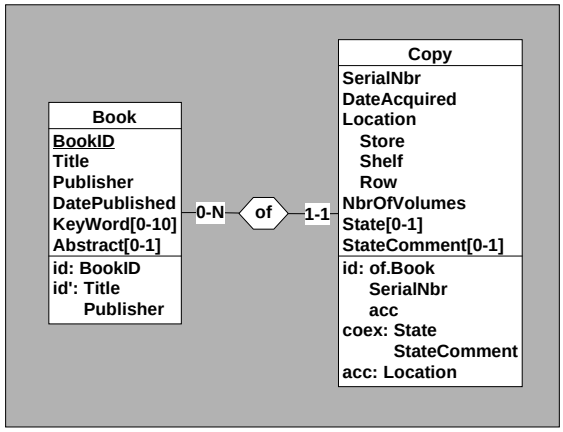


Figure 2-6: Some constraints. BookID is a primary identifier and {Title, Publisher} a secondary identifier of Book. SerialNbr identifies each Copy within a definite Book. In addition, this identifier is an access key. Optional attributes State and StateComment both are valued or void (coexistence).

Inter-group integrity constraint

Independently of their function(s), two groups can be related through a relation that expresses an *inter-group integrity constraint*.

The following constraints are available:

- *Reference:* the referencing group references the referenced group. The referenced group (the target of the constraint) has to be an identifier (primary or secondary) of its parent entity type. The referencing group (the origin of the constraint) should have the same structure (same length and same type for all the corresponding components) as the referenced group. The values of the components of the referencing group in an entity identify an entity of the referenced entity type. In Figure 2-7, the entity type Copy contains a reference group, tagged with ref, made up of one attribute which references the entity type Book. The attributes BookID of Copy and Book are both strings of the same length.
- *Equality:* this is a special kind of reference constraint in which every entity of the referenced type must be referenced as well. Graphically, the sole difference is the tag that becomes equ.
- *Inclusion*: it is a generalization of the reference constraint in which the target group does not need to be an identifier; it shows that every instance value of the origin group must be an instance value of the target group.
- *Generic inter-group constraint*: can be drawn from any group to any other group of the schema; defining the semantics of this constraints is up to the designer.

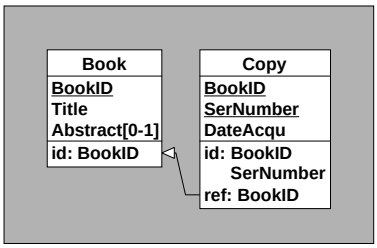


Figure 2-7: Attribute BookID form a reference group (foreign key) to Book.

2.2.6 Processing Unit

A processing unit is any dynamic or logical component of the described system that can be associated with a schema, an entity type or a relationship type. For instance, a process, a stored procedure, a program, a trigger, a business rule or a method can each be represented by a processing unit.

There are four types of anchored processing units:

- *Method*: service which the object class is responsible for; used in advanced ER and OO models; can represent functions of abstract data types too.
- *Predicate*: logical rule stating a time-independent property.
- *Trigger*: active rule.
- *Procedure*: any other kind of processing units.

2.2.7 Collection

A *collection* is a repository for entities. A collection can comprise entities from different entity types, and the entities of a given type can be stored in several collections. Though this concept can be given different interpretations at different levels of abstraction, it will most often be used in physical schemas to represent files, data stores, table spaces, etc. In Figure 2-8, the collection Library represents a file in a library management system. It stores the rows of the tables Book, Author and Copy.

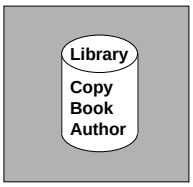


Figure 2-8: An example of collection.

2.2.8 Dynamic Properties

Each concept presented above has a series of properties, such as the name of an entity type or of an attribute, the cardinalities of an attribute or a role, the total and disjoint property of an ISA relation, etc. A *dynamic property* is a user-defined property that can be dynamically appended to every concept. This property can be of various types: integer, character, boolean, real or string. It can also be mono-valued or multi-valued.

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. Concentrating on federation systems, especially the mediation layer, [Busse, 1999] distinguishes the following kinds of meta-properties specific to schema integration:

- *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.

2.3 Model Specialization

This generic model can be *specialized* into any data model. A specialized model is 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 2-9 shows some common interpretation of the generic constructs.

As an illustration, 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. There is no rel-types.
- *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, ISA 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 2-9: Some common interpretations of the generic concepts.

2.4 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.

2.4.1 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 this thesis 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 2-10). The most important one concerns multivalued attributes, which can be represented trough array attributes only. In addition, optional attributes are not explicitly represented except as multi-valued attributes.

An example of a COBOL schema is shown in Figure 2-11. In this model, record types have (alternate) record keys (e.g. Customer has one attribute that plays the role of identifier and access key). Attributes are atomic or compound (e.g. Cust-Address) or multivalued (e.g. Cust-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	*... occurs N times*	Mandatory
Identifier + access key	Record key, alternate record key	Only one attribute
Non-identifier access key	Alternate record key *with duplicates*	Only one attribute
Collection	Files	

Figure 2-10: Concepts and constraints of the COBOL data model.

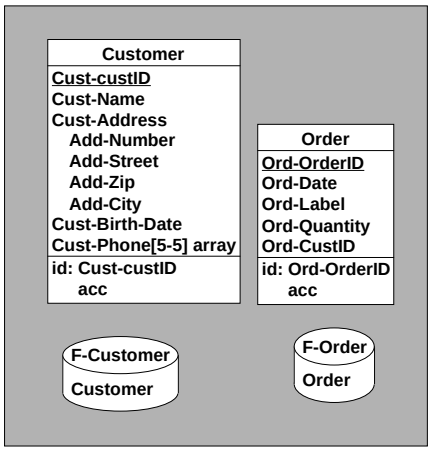


Figure 2-11: Cobol structure example.

Relational data model

The attribute structure in the relational data model is particularly poor (Figure 2-12): an attribute must be single-valued and atomic. It can nevertheless be optional. Moreover, the relation 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	*Nullable* 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	

Figure 2-12: Concepts and constraints of the relational data model.

An example of a relational data schema is shown in Figure 2-13. In this model, tables have primary and unique keys (e.g. Customer has a primary identifier). All the attributes are atomic and monovalued. 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 at-

tribute of Customer). Names are formed according to the relational language syntax.

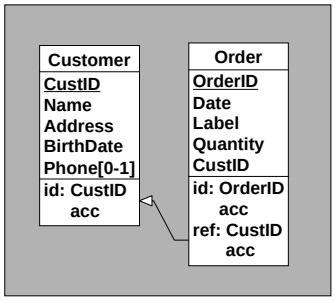


Figure 2-13: Relational schema example.

2.4.2 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 this thesis, we propose two categories of canonical data models; namely the *binary conceptual model* and the *operational models*. Such categories reflect the distinction between theoretical and operational aspects of database federations.

- The *binary conceptual model* is based on ER model[1] supporting binary, non cyclic relationships and generalization hierarchies. This model will be used to illustrate the theoretical aspects of the mapping definition (Chapter 3) and of the schema-oriented framework of the proposed federation system (Chapters 4 and 6).
- The *operational models* are those that are explicitly implemented by the wrapper prototypes presented in Chapter 5. We propose two operational models: the *wrapper logical model* associated with a common data manipulation language close to SQL and the *wrapper object-oriented model* that offers Java object methods for accessing read-only data.

The rest of this section overviews the features of each of the operational models.

Wrapper logical model

The *Wrapper logical Data Model* (WDM) 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 WDM as a model that includes all the

1. We choose the ER model instead of the UML model because of its expressive power and its formalization. For a comparison between these two models, we refer to [Hainaut, 2002b].

structures and constraints that exist explicitly in these two data models (Figure 2-14). As a result, WDM comprises entity types, attributes (that can be mono- or multi-valuated; simple 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 2-15. In this model, entity types can 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 can be mandatory (e.g. name attribute of Client) or optional (e.g. birth-date attribute of Customer). Attributes can also have several values (e.g. phone attribute of Customer). Entity types can 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 2-14: Logical wrapper data model: constructs and constraints.

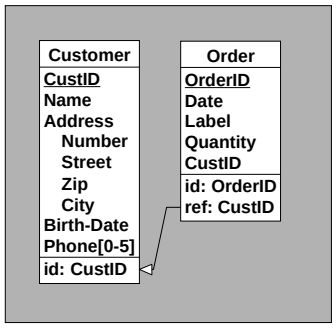


Figure 2-15: Wrapper logical schema example.

Object-oriented model

The *wrapper object-oriented model* (Figure 2-16) 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.

An *object type* (named entity type in the generic model) definition is a Java class 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 2-16: 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 2-15).

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 2-15, 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 attributes: 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 that refers its source entity object. The property defined on a one-to-one relationship is get*EntityObjectName* whereas the properties defined on a many-to-one relationship are getFirst*EntityObjectName* and getNext*EntityObjectName*.

The translation between the wrapper logical and the wrapper object-oriented models is discussed in [Noël, 2001].

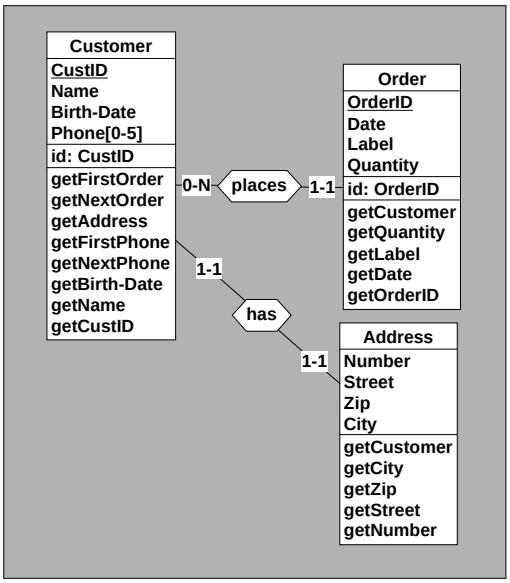


Figure 2-17: Wrapper object-oriented schema example.

Object identifier

To achieve full transparency towards applications accessing the legacy database through the object wrapper, *object identifiers* (oids) must be provided for such objects. Oids for objects should have exactly the same properties they have in a regular object-oriented environment. In particular, an oid must identify the same object each time it used in a method evocation. Satisfying such requirements in our setting is complicated. Legacy databases, such as relational databases, provide only value-based identity, and values on which an oid might be bases (such as primary key values) may be changed without notice due to the update autonomy of the underlying database. Object wrappers are equipped with local identity monitors,

which use mechanisms such as locking or caching of objects to simulate object identity even in presence of updates on the corresponding relational tuples. Dealing with oids is out of scope of this thesis however. Strategies for implementing oids in wrappers have been developed in [Hohenstein, 1996] and [Lim, 1999].

Chapter 3

Mapping Definition

In which the mappings between two schemas are defined as schema transformation operators. The concept of schema transformations is presented and illustrated. We explain how schema transformations can be used to automatically translate queries. We then introduce the notion of history by giving its definition and its properties. We finally present the model translation concept to illustrate those of schema transformation, query transformation and transformation history.

3.1 Introduction

It can be shown that mappings can be modeled by 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 DMS schema translation [Hainaut, 1993b; Rosenthal, 1988; Rosenthal, 1994] or data conversion [Navathe, 1980]. As will be shown later on, they can be used for wrapping engineering of legacy databases and integration as well.

3.2 Mapping Baselines

Current mapping definitions of wrappers and mediators, such as TSIMMIS [Chawathe,

1994], InterViso [Templeton, 1995], IM [Levy, 1996], Garlic [Roth, 1997] and Clio [Yan, 2001] are what can 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 thesis 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]:

- Decomposition of schema transformation into a sequence of small steps, whereas the query-processing oriented approaches require that constructs in one schema are directly defined in term of those in the other schema (Sections 3.3 and 3.4).
- Using the schema transformations to automatically translate queries on the global schema into queries on the local schemas (Section 3.5).
- Enabling software production automation: as the transformation can be completely formalized, they can be implemented in a CASE tool (Chapters 5 and 7).
- Providing a unifying basis between the software production and the methodology (Chapters 4 and 6).

As already stated, the generic data model defined in Chapter 2 is the ideal support for schema transformations. 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 data model brings several important benefits:

- A transformation can be carried out for a construct of a data model M_1 where the result of this transformation is defined in terms of another data model M_2. This allows inter-model transformations to be applied, where the constructs of one data model are replaced with those of another.
- These inter-model transformations form the basis for semi-automated generation of the legacy data wrappers presented in Chapter 5.

3.3 Schema Transformation

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).

A transformation Σ can be completely defined by a pair of mappings <T,t> (Figure 3-1):

- T is called the *structural mapping*. It replaces source construct C in schema S with construct C'; C' is the target of C through T and is noted C'=T(C). In fact, C and C' are classes of constructs that can be defined by structural predicates. T is therefore defined by a minimal precondition P that any construct C must satisfy in order to be transformed by T, and a maximal postcondition Q that T(C) satisfies. T is the syntax of the transformation.
- t is the *instance mapping* that states how to produce the T(C) instance that corresponds to any instance of C. If c is the instance of C, then c'=t(c) is the corresponding instance of T(C). t is the semantics of the transformation.

Another equivalent way to describe mapping T consists of a pair of predicates <P,Q>, 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 $\Sigma \equiv$ <P,Q,t>.

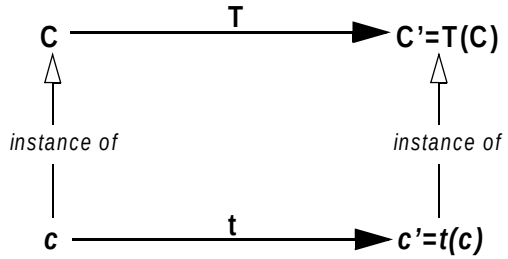


Figure 3-1: The two mappings of schema transformation $\Sigma \equiv$ <T,t>.

3.3.1 Reversibility

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

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

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

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

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* (SR-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,

$$P_1(C) \Rightarrow \quad (C = T_2(T_1(C))) \wedge (c = t_2(t_1(c)))$$
$$P_2(C) \Rightarrow \quad (C = T_1(T_2(C))) \wedge (c = t_1(t_2(c)))$$

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 $<P_1,Q_1,t_1,t_2>$. 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.

3.3.2 Structural Analysis of a Transformation

A transformation is known to replace construct C with construct C' in schema S, to yield new schema S'. The effect of a transformation T in schema S can be specified as follows. We define a schema S as a set of constructs. 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 other schemas.

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

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

We also have:

$$C_S(T) = C_0(T) \cup C_-(T)$$
$$C_{S'}(T) = C_0(T) \cup C_+(T)$$
$$S' = (S - C_-(T)) \cup C_+(T)$$

These concepts are illustrated in the following scenario, in which an instance of the rel-type/ entity type transformation of Figure 3-2 is applied on rel-type R, and in which every construct has been given a denotation:

$C_-(T)$ = {R, rA, rB}

$C_+(T)$ = {R', R1, R2, rR1A, rR1R', rR2B, rR2R', id(R')}

$C_0(T)$ = {A, B}

S = {A, B, R, rA, rB}

S' = {A, B, R', R1, R2, rR1A, rR1R', rR2B, rR2R', id(R')}

$C_S(T)$ = {A, B, R, rA, rB}

$C_{S'}(T)$= {A, B, R', R1, R2, rR1A, rR1R', rR2B, rR2R', id(R')}

3.3.3 Signature of a Transformation

A transformation is specified through its *signature*, that states the name of the transformation, the names of the concerned constructs in the source schema, and the names of the new constructs in the target schema. For example, the signature of the transformations T_1 and T_2 in Figure 3-2 are as follows[1]:

T_1: (R',{(A,R1),(B,R2)}) ← RT-ET(R)

T_2: R ← ET-RT(R')

The first one is interpreted as *"when applying RT-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-RT to entity type R', the new rel-type is called R"*.

The constructs 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 T_2 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 C_0 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 T_1 and T_2 described above.

1. Fixed-length lists of a signature are enclosed into parentheses, while variable-length lists are enclosed into curly brackets.

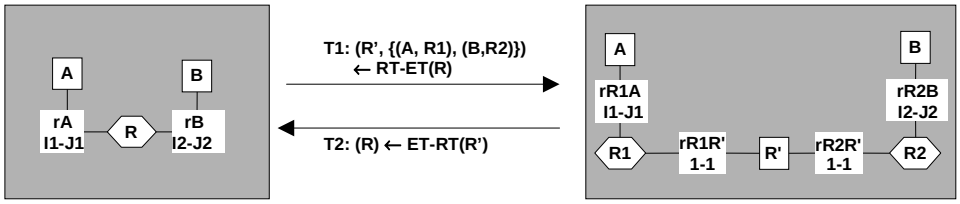


Figure 3-2: Representation of structural mapping T_1 (from left to right) and T_2 (from right to left) of a typical SR-transformations. The identifier of the entity type R' is made up of the roles rR1R' and rR2R'.

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

(R',{(A,R1),(B,R2)}) ← RT-ET(R)

could be instantiated, in the actual schema shown in Figure 3-3, into:

(Order,{(Customer,passes),(Product,of)}) ← RT-ET(Orders)

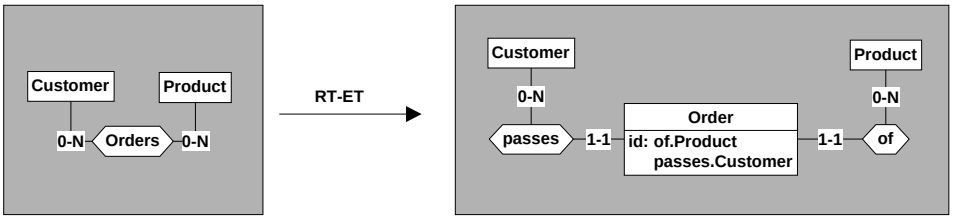


Figure 3-3: An instantiated transformation.

From these examples, we can observe an essential property of the signatures: their *reversibility*. Being provided with the right-side schema of Figure 3-2 and the signature of T_2, we can derive the signature of T_1, and conversely. In other words, the signature provides enough information, not only for *redoing* the operation, but also to *undo* it.

This property is less obvious for some non-SR-transformations. Let us consider the example of the del-ET operator, which removes an entity type from a schema. It can be illustrated as follows. At first glance, it seems that the following signature could be quite right:

() ← del-ET(B)

Unfortunately, the problem is that, though we can redo the transformation, we are unable to undo it. Of course, we are informed that entity type B was removed, but we have lost information about its structure: what were its attributes, its roles, its constraints, etc.?

In this case, we must augment the signature with those of the derived operations. We consider that removing B consists in removing its constraints (e.g. identifiers), then its attributes and its roles, then the inconsistent relationship types, and finally B itself:

() ← del-ID(B,{B1},δ_1)
() ← del-Att(B,B1,δ_2)
() ← del-Att(B,B2,δ_3)
() ← del-Role(R,B,δ_4)
() ← del-Role(R,A,δ_5)
() ← del-RT(R,δ_6)
() ← del-ET(B,δ_7)

In these signatures, the symbol δ_i stands for any kind of additional information needed to create the construct, e.g. value type, value length, cardinality constraint, narrative description, etc. Now the signature of the del-ET operation is reversible, though the operation itself is not.

3.3.4 Schema Transformation Sequence

A *transformation sequence* is a sequence of $n \geq 1$ primitive transformations: $T_p = <T_1\ T_2\ ...\ T_n>$. For instance, $T_p = <T_1\ T_2>$ is obtained by applying T_2 on the schema that results from the application of T_1. As for primitive transformations, if a schema S can be transformed to a schema S' by means of semantics-preserving transformations, and vice-versa, then S and S' are equivalent.

As an illustration, Figure 3-4 shows a sequence of three transformations usually used in database engineering process. The first one (T_1) replaces a foreign key with a relationship type, the second one (T_2) expresses a multiple attribute as an external entity type, and the third one (T_3) renames the name of an entity type.

Practically, this concept of schema transformation sequence has very diverse applications. For example, if we wish to establish the mappings between two predefined schemas, the transformational approach consists in finding a chain of transformations which, applied to the source schema, produces the target schema. The underlying chain of structural mappings reflects the structural mappings between the two schemas whereas the chain of instance mappings reflects the correspondence at the data level themselves. If each transformation of the chain of transformations is symmetrically reversible, then the two schemas are semantically equivalent.

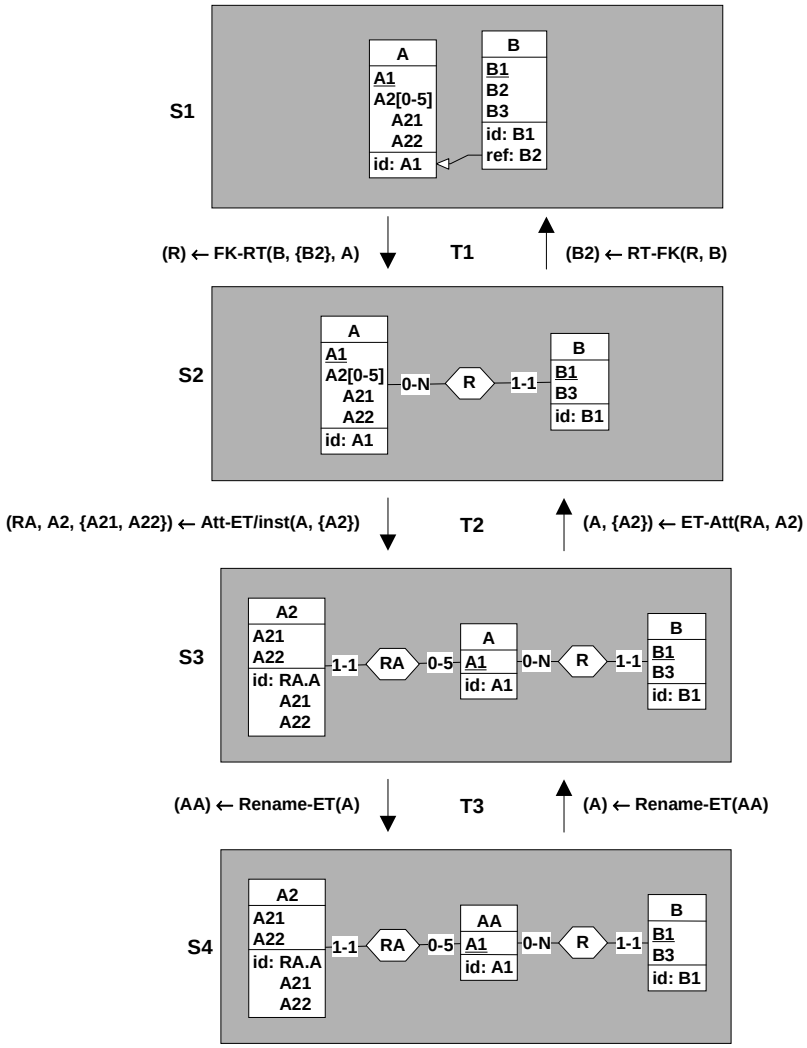
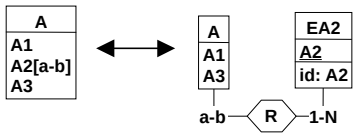
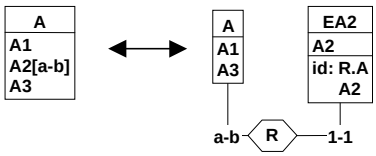
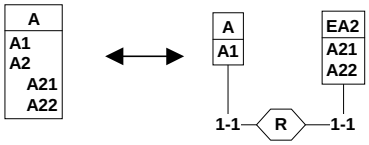


Figure 3-4: Sequence of three common (semantics-preserving) schema transformations: a foreign key transformation followed by an attribute transformation and a renaming transformation.

3.4 Some Typical Transformations

We propose in Figure 3-5 and Figure 3-6 two sets of the most commonly used transforma-

tional operators. The first one is sufficient to carry out the transformation of most conceptual schemas into relational schemas. The second comprises transformations that we will use for the wrapper development. 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 paradigms[2].

Att-ET/val: Transforming an attribute into an entity type (value representation). *Inverse*: **ET-Att**.	 (EA2, {A2}) ← **Att-ET/val**(A, {A2}) *(direct)* (A, {A2}) ← **ET-Att**(EA2) *(inverse)*
Att-ET/inst: Transforming an attribute into an entity type (instance representation). *Inverse*: **ET-Att**.	 (EA2, {A2}) ← **Att-ET/inst**(A, {A2}) *(direct)* (A, A2) ← **ET-Att**(EA2) *(inverse)*
Disagg: Disaggregating a compound attribute. *Inverse*: **Aggreg**	 (EA2, {A21, A22}) ← **Disagg**(A, {A2}) *(direct)* (A, A2) ← **Aggreg**(EA2) *(inverse)*

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

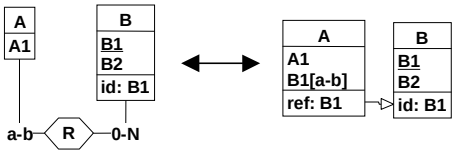
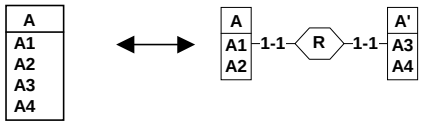
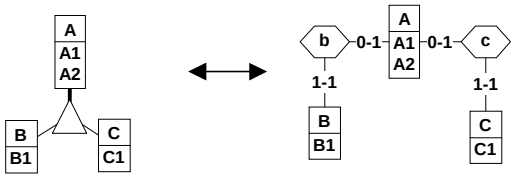
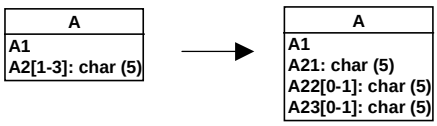
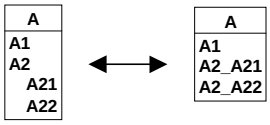
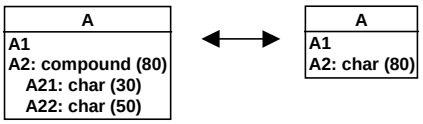
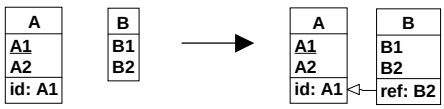
RT-FK: Transforming a binary relationship type into a foreign key. *Inverse*: **FK-RT**.	 (A, {B1}) ← **RT-FK**(R) *(direct)* (R) ← **FK-RT**(A, {B1}) *(inverse)*
Split: Splitting an entity type into two entity types. *Inverse*: **Merge**.	 (A') ← **Split**(A, {A3, A4}) *(direct)* (A, {A3, A4}) ← **Merge**(A') *(inverse)*
ISA-RT: Materializing an ISA relation into relationship type. *Inverse*: **RT-ISA**	 (b, c) ← **ISA-RT**(A, B, C) *(direct)* (A, A2) ← **RT-ISA**(A) *(inverse)*

Figure 3-5: Six major generic transformations with their inverse. Cardinalities a, b, c and d must be replaced with actual values.

MultAtt-Serial: Replacing a multivalued attribute with a series of single-valued attributes that represents its instances. *Inverse*: **Serial-MultAtt.**	 (A, {A21, A22, A23}) ← **MultAtt-Serial**(A, {A2}) *(direct)* (A, {A2}) ← **Serial-MultAtt**(A, {A21, A22, A23} *(inverse)*

MultAtt-Single: Replacing a multivalued attribute with a single-valued attribute that represents the concatenation of its instances. *Inverse*: **Single-MultAtt**.	 (A, {A2}) ← **MultAtt-Single**(A, {A2}) *(direct)* (A, {A2}) ← **Single-MultAtt**(A, {A2}) *(inverse)*
CompAtt-Serial: Replacing a compound attribute with a series of atomic attribute that represents its component attributes. *Inverse*: **Serial-CompAtt**.	 (A, {A2_A21, A2_A22}) ← **CompAtt-Serial**(A, {A2}) *(direct)* (A, {A2}) ← **Serial-CompAtt**(A, {A2_A21, A2_A22}) *(inverse)*
CompAtt-Single: Replacing a compound attribute with an atomic attribute that represents the aggregation of its component attributes. *Inverse*: **Single-CompAtt**.	 (A, {A2}) ← **CompAtt-Single**(A, {A2}) *(direct)* (A, {A2}, {A21, A22}) ← **Single-CompAtt**(A, {A2}) *(inverse)*
Create-Reference: A reference constraint is added. The referencing group and the group it references are made up of existing attributes. and/or roles. *Inverse*: **Del-Reference**.	 () ← **Create-Reference**(B, {B2}, A, {A1}) *(direct)* () ← **Del-Reference**(B, {B2}, δ) *(inverse)*

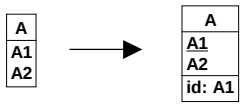
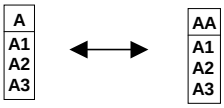
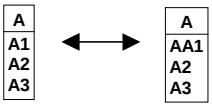
Create-Identifier: An identifier group is added. The group is made up of existing attributes and/or roles. *Inverse*: **Del-Identifier**.	
	() ← **Create-Identifier**(A, {A1}) *(direct)* () ← **Del-Identifier**(A, {A1}, δ) *(inverse)*
Rename-ET: An entity type is renamed. *Inverse*: **Rename-ET**	
	(AA) ← **Rename-ET**(A) *(direct)* (A) ← **Rename-ET**(AA) *(inverse)*
Rename-Att: An attribute is renamed. *Inverse*: **Rename-Att**	
	(A, {AA1}) ← **Rename-Att**(A, {A1}) *(direct)* (A, {A1}) ← **Rename-Att**(A, {AA1}) *(inverse)*

Figure 3-6: Generic transformations commonly used in database wrapping.

Besides the generic transformations described above, we introduce a special kind of transformations: Create-FunctionGroup. This transformation defines that the components of the added group are calculated by using a *user-defined function* F whose semantics is known by the user only. It has no predefined t mapping part, so that the instance conversion cannot be predefined, but must be manually written. They are used as conversion functions for the attributes or as reconciliation functions for resolving conflicts that have not been formally defined by other schema transformations. The conversion of Belgian Francs to Euros is such an example. In Chapter 6, we will develop the applications of these functions.

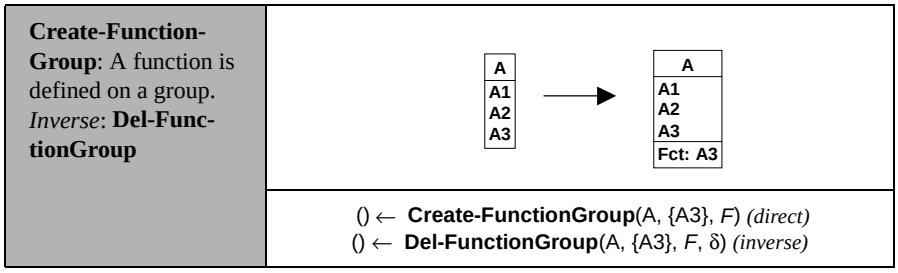


Figure 3-7: Group Transformation associated to a user-defined function F.

3.5 Schema and Query Mapping

In this section, we show how the schema transformation sequences can be used to automatically translate queries in either direction between a pair of schemas. More precisely, for a schema transformation sequence between two schemas S_1 and S_2, we will see how this sequence can be used to automatically:

- Translate queries posed on S_2 to queries posed on S_1;
- Translate updates posed on S_2 to updates posed on S_1.

So far, we have considered the transformation of one source schema to one target schema. In this section, we extend our approach to schema integration. We show in this way how a global query can be translated into local queries using a schema transformation approach.

3.5.1 Model and Query Language

For simplicity and clarity, we consider the generic binary model presented in Chapter 2. This language is compliant with standard files, SQL-2 and ER models. It is enough expressive and generic to describe all the main structures and constraints that exist explicitly in these data models:

- Attributes can be atomic or compound; single-valued or multivalued;
- Reference, identifier and access groups;
- Entity types with at least an attribute and an identifier;
- Binary, non cyclic relationship types, without attribute;
- ISA relations.

We provide a simple query language that is homogeneous with the generic binary model: que-

ries in Query are conjunctions of schema constructs. A query answer is an instance set of schema constructs.

A query Query over a schema S is an expression whose variables are constructs of S. Query is a query expression over the schema which defines the instance of constructs. The syntax of a Query is:

```
Query ::=     Schema | Predicate | [or, Query, Query {, Query}] |
              [and, Query, Query {, Query}] | [not, Query]
Predicate ::= [eq, Atom, Atom] | [less, Atom, Atom]
```

Schema identifies a schema construct. This is the construct being added or deleted by a transformation. In other words, this is one of the constructs that take part in the definition of a schema transformation signature. Schema includes variable(s) used to instantiate instances of the construct and it takes one of the forms presented in Figure 3-8. The identifiers in bold are literal, identifiers in italic are constants; other identifiers (neither in bold nor in italic) are variables; and the underscore character is an anonymous variable. Atom represents a variable declared in a schema construct.

When eq refers two variables of the same query, we can simplify the query and omit this predicate, e.g. we need only write [**att**, *Person, Id,* EP, 4] instead of [**and**, [**att**, *Person, Id,* EP, ID], [**eq**, [ID, 4]].

Construct	Syntax	Semantics
Entity type	[**ent**, Name, Et]	represents an entity type called Name, and Et can be instantiated with instances of Name
Attribute	[**att**, OwnerName, AttName, Owner, Att]	represents an attribute AttName of a construct OwnerName. The type of OwnerName can be either an entity type or a compound attribute. OwnerName contains the name(s) of the parent(s) of the attribute. Att can be instantiate with a value of the attribute associated with the instance Owner of OwnerName
Relationship type	[**rel**, ET1Name, RTName, ET2Name, ET1, ET2]	represents a relationship RTName between entities ET1Name and ET2Name. ET1 and ET2 can be instantiate with entity instances involved in the relationship

Figure 3-8: Syntax and semantics of the main constructs of the generic data model.

Example

In Figure 3-9, we illustrate the main representations of the constructs depicted in Figure 3-8.

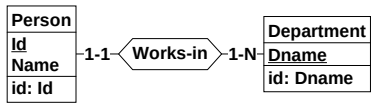
Graphical Representation	Query Language Representation	
	Construct	**Example**
	Entity type	[**ent**, *Person*, EP]
	Attribute	[**att**, *Person, Id,* EP, ID]
	Relationship type	[**rel**, *Person, Works-in, Department,* EP, ED]

Figure 3-9: Examples of construct representation (EP, ED and ID represent variables).

3.5.2 Schema Transformation and Query Substitution

Let us assume a schema S_1 is transformed into a schema S_2 and the queries posed on S_1 have to be translated to queries posed on S_2. Consider first the case where S_1 is transformed into S_2 by a single primitive transformation T. The only cases we need to consider in order to translate a query Q_1 posed on S_1 to an equivalent query Q_2 on S_2 are to apply renamings and to substitute occurrences of $C_{\_}(T)$ (Figure 3-10). For transformation sequences, the substitutions are successively applied in order to obtain the final query Q_2.

Transformation	Signature	Substitution
RenameET	(name') ← **RenameET**(name)	Q_2 = [name' / name] Q_1
RenameAtt	(name') ← **RenameAtt**(ET, name)	Q_2 = [ET, name' / ET, name] Q_1
Other	(S_2) ← **T**(S_1)	Q_2 = [$C_{\_}(T)$ / query] Q_1
Other *(inverse)*	(S_2) ← $\mathbf{T}^{-1}(S_1)$	Q_1 = [$C_{\_}(T^{-1})$ / query] Q_2

Figure 3-10: Schema transformation and query substitution.

Let us consider now an update U_1 posed on S_1, taking one of the general forms where ET is an entity type of S_1, RT is a relationship type of S_1 and $query_{att}$ is a query limited to conjunctions of attribute variables:

insert ET $query_{att}$
delete ET $query_{att}$
insert RT $query_{att}$
delete RT $query_{att}$

Then exactly the same substitutions as for queries above can be applied to U_1 in order to ob-

tain an equivalent update U_2 posed on S_2. Note that U_2 is unambiguous only if S_1 is an updatable view of S_2, otherwise all the usual problems associated with view updates will need to be addressed ([Bencilhon, 1981], [Dayal, 1981]).

We illustrate these notions by first giving an example of query translation between two equivalent schemas. We next show how we can translate these queries between two non-equivalent schemas. We will later give a larger example within the framework of the model transformation example presented in Section 3.7.

Single primitive transformation between equivalent schemas

We illustrate a query transformation for a single primitive schema transformation T1 and its inverse $T1^{-1}$ between the pair of schemas S_1 and S_2 illustrated in Figure 3-12. Let us consider the structural functions C_- and C_+ of T1:

$C_-(T1) = C_+(T1^{-1}) = \{DName\}$

$C_+(T1) = C_-(T1^{-1}) = \{works\text{-}in, Department, Dname\}$

The schema transformation T1 and its inverse $T1^{-1}$ are defined in Figure 3-11 by means of:

- their signature;
- their structural function C_- expressed in the schema form;
- the queries query that state how the extents of each constructs of C_- can be recovered from the extents of the remaining schema constructs C'.

T1	(Department, {Dname}) ← Att-ET/val(Person, {Dname})
C_-(T1)	**Query**
[**att**, *Person, Dname,* ED, X]	[**and** [**rel**, *Person, Works-in, Department,* EP, ED], [**att**, *Department, Dname,* ED, X]]

$T1^{-1}$	(Person, {Dname}) ← ET-Att(Department)
$C_-(T1^{-1})$	**Query**
[**att**, *Department, Dname,* ED, X]	[**att**, *Person, Dname,* EP, X]
[**ent**, *Department*, ED]	[**att**, *Person, Dname,* _, ED]
[**rel**, *Person, Works-in, Department,* EP, ED]	[**att**, *Person, Dname,* EP, ED]

Figure 3-11: Example of schema transformation signatures and the queries that state how the extents of each construct of C_- can be recovered from the extents of the remaining schema constructs C'(T1).

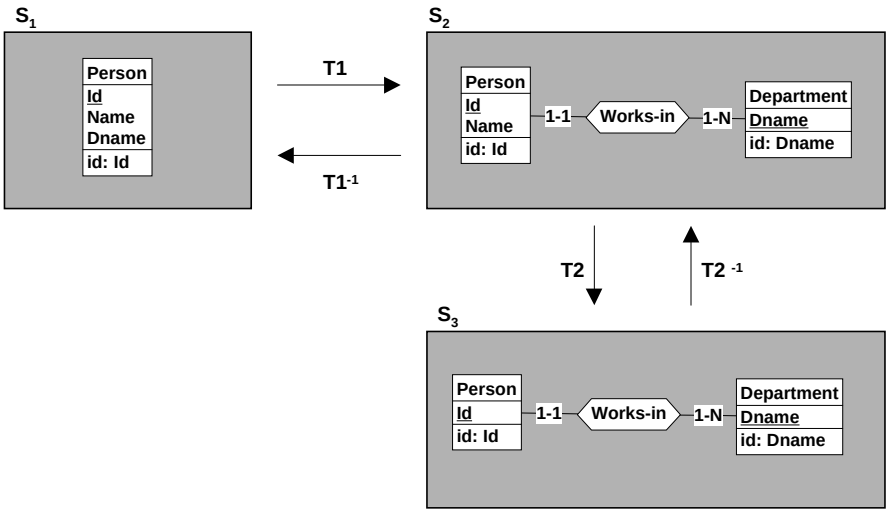


Figure 3-12: Example of schema transformations and their inverse.

For any query on S_1, the transformation definition can be used to translate constructs of S_1 into constructs of S_2, resulting in a query on S_2. In that way, Figure 3-13 shows how the request *"Find the ids of all the persons that work in the department of computing"* is translated into equivalent queries in S_1 and in S_2 by applying the above substitution.

Query on S_1	**Query on** S_2
[**and**, [**att**, *Person*, *Id*, EP, ID], [**att**, *Person, Dname,* EP, "Computing"]]	[**and**, [**att**, *Person, Id,* EP, ID], [**rel**, *Person, Works-in, Department,* EP, ED], [**att**, *Department, Dname,* ED, "Computing"]]

Figure 3-13: Example of query translation from S_1 to S_2.

Figure 3-14 shows the translation of an insert query from S_2 to S_1. The insertion on S_2 of a new person with id 10, name "Michaux" that works in the department "Poetry", can be translated into an insertion on S_1 of a person with id 10, name "Michaux" and Dname "Poetry".

Update on S_2	Update on S_1

insert
[**and**,
  [**ent**, *Person*, EP],
  [**att**, *Person, Id,* EP, 10],
  [**att**, *Person*, *Name*, EP, "Michaux"],
  [**rel**, *Person*, *Works-in, Department,* EP, ED],
  [**att**, *Department, Dname,* ED, "Poetry"]
]
	insert
	[**and**,
	  [**ent**, *Person*, EP],
	  [**att**, *Person, Id,* EP, 10],
	  [**att**, *Person, Name,* EP, "Michaux"],
	  [**att**, *Person, Dname,* EP, "Poetry"]
	]

Figure 3-14: Example of update translation from S_2 to S_1.

Transformation sequence between two non equivalent schemas

Data, updates and queries can be translated between a pair of non-equivalent schemas if these use their common constructs. If a query uses some construct which has no derivation in the target schema, as for example in Figure 3-15 below, then the result query will contain void term since the attribute Name has been deleted by applying the transformation T2.

Query on S_1	Query on S_3

[**and**,
  [**att**, *Person, Id,* EP, ID],
  [**att**, *Person, Name,* EP, "Michaux"],
  [**att**, *Person, Dname,* EP, "Poetry"]
]
	[**and**,
	  [**att**, *Person, Id,* EP, ID],
	  [**void**],
	  [**rel**, *Person, Works-in, Department,* EP, ED],
	  [**att**, *Department, Dname,* ED, "Poetry"]
	]

Figure 3-15: Example of an invalid query translation from S_1 to S_3.

Query simplification

Queries that result from a query substitution can be sometimes simplified into an equivalent query that is more efficient to execute but gives the same result as the original one. This concerns an *n-entity query* q that can be transformed into an equivalent (n-1)-entity query q'.

Let us illustrate the principle by using the schema transformation RT-FK between the schema S_2 and S_6 illustrated in Figure 3-16. Assume the query posed on S_2 that returns the persons who work in the poetry department (Figure 3-17). By applying the transformation RT-FK, we obtain a correct query posed on S_6 (at the bottom and the left of Figure 3-17). This query is a n-entity query in that it involves two entity types. This n-entity query can be simplified by transforming it into a mono-entity query because of the reference constraint group defined on the attribute Dname of Person. The resulting query is shown at the bottom and the right of Figure 3-17.

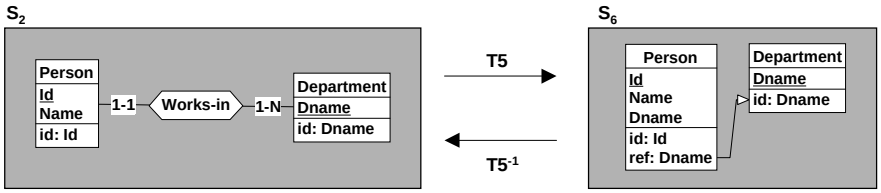


Figure 3-16: Schema S_6 that results of schema transformation RT-FK on S_2 of Figure 3-12.

Query on S_2
[**and**, [**att**, *Person, Id,* EP, ID], [**rel**, *Person*, *Works-in, Department,* EP, ED], [**att**, *Department, Dname,* ED, "Poetry"]]

Multi-entity Query on S_6	**Simplified Query on** S_6
[**and**, [**att**, *Person, Id,* EP, ID], [**att**, *Person, Dname,* EP, X], [**att**, *Department, Dname,* ED, X], [**att**, *Department, Dname,* ED, "Poetry"]]	[**and**, [**att**, *Person, Id,* EP, ID], [**att**, *Person, Dname*, EP, "Poetry"]]

Figure 3-17: Non-simplified and simplified queries on S_6.The non-simplified query results from the query mapping based on the RT-FK transformation.

There are many rules for simplifying n-entity queries into equivalent (n-1)-entity queries. We refer to [Johnson, 1984] for a discussion about query simplification[3]. Here, we only present the most representative one: this resulting from a RT-FK transformation. The query equivalence is guarantee by the reference constraint. For simplicity, we assume that the reference group is made up of only one attribute.

Given a query q that represents n-entity types ET_i ($1 \leq i \leq n$), the query simplification proceeds as follows: First, an ET_i in q that references no other ET_j in q ($i \neq j$) is chosen. Let ET_{target} be the chosen entity type. Then, query q is subdivided into 2 subqueries $[q_1; q_2(ET_{target})]$ where $q_2(ET_{target})$ only represents schema constructs that have ET_{target} as variable. If each schema construct of $q_2(ET_{target})$ refers to the identifier[4] of ET_{target}, then the query q can be simplified: the instances of schema construct that references ET_{target} are obtained by taking

3. Note that [Johnson, 1984] only considers the conjunctive queries.
4. A referenced attribute is an identifier (Chapter 2, Section 2.2.5).

the values of the identifier of ET_{target}. The rule works recursively until there remain no subqueries to be reduced for any ET_{target}.

Example

To illustrate this rule, we apply it to the non-simplified query of Figure 3-17. First, the entity type Department is chosen as ET_{target}. The query $q(ET_{target})$ is then extracted from q with respect to the rule described above:

$q(ET_{target})$:[**and**,
[**att**, *Department, Dname,* ED, X],
[**att**, *Department, Dname,* ED, "Poetry"]]

$q(ET_{target})$ only represents the identifier of Department; q is then simplified. The instances of the attribute Dname of Person (the variable X) are obtained by taking those instances of Dname of Department with value "Poetry".

q: [**and**,
[**att**, *Person, Id,* EP, ID],
[**att**, *Person, Dname*, EP, "Poetry"]]

3.5.3 Schema Integration and Queries

Data integration approaches

Data integration is a process by which several databases, which associated local schemas, are integrated to form a single virtual database with an associated global schema. Up to now ([Cali, 2001], [Lenzerini, 2001]) data integration approaches have been either *Global as View* (GaV) or *Local as View* (LaV):

- In the GaV approach, the constructs of a global schema are described as views over the global schemas. These view definitions are used to rewrite queries over a global schema into distributed queries over the local schemas.
- In the LaV approach, the constructs of the local schemas are defined as views over the global schema, and processing queries over the global schema involves rewriting queries using views.

The principal drawback of GaV is that it does not readily support the evolution of local schemas. On the other hand, LaV isolates the changes to local schemas to impact only on the derivation rules defined for that schema. However, LaV has problems if one needs to change the global schema, since all the rules for defining local schemas as views of the global schema need to be reviewed.

Examples of the GaV approach are Tsimmis [Chawathe, 1994], InterViso [Templeton, 1995] and Garlic [Roth, 1997] while an example of the LaV approach is IM [Levy, 1996].

Reversible schema transformation

The schema transformation approach offers an unifying framework for GaV and LaV approaches, based on the reversibility of schema transformations (Section 3.3.1). Using reversible schema transformations, it is possible to extract definitions of the global schema as a view over the local schema, and it is also possible to extract definitions of the local schemas as views over the global schema. An interesting discussion on the use of schema transformations in the LaV and GaV approaches can be found in [McBrien, 2003].

Schema and query translation

In Figure 3-18, schema S_5 can be regarded as a global schema which contains the union of the information of the two source schemas S_3 and S_4. Since S_5 contains constructs which cannot be derived from all the schemas (for instance, the attribute Name of Person is not hold in S_4), there will be some global queries on S_5 which will result in void.

To illustrate this, Figure 3-19 below first shows how a global query on S_5 would be translated to each of the two source schemas (GaV approach). It then shows how these two queries can be integrated into a global query plan. The construct [source, schema, query] identifies that a sub-query can be sent to a particular source. If more than one source contains the information, the various alternative queries are placed in construct [plan, $source_1$,..., $source_N$], indicating that some further query planning is required in order to choose one of the queries for execution or to form the result.

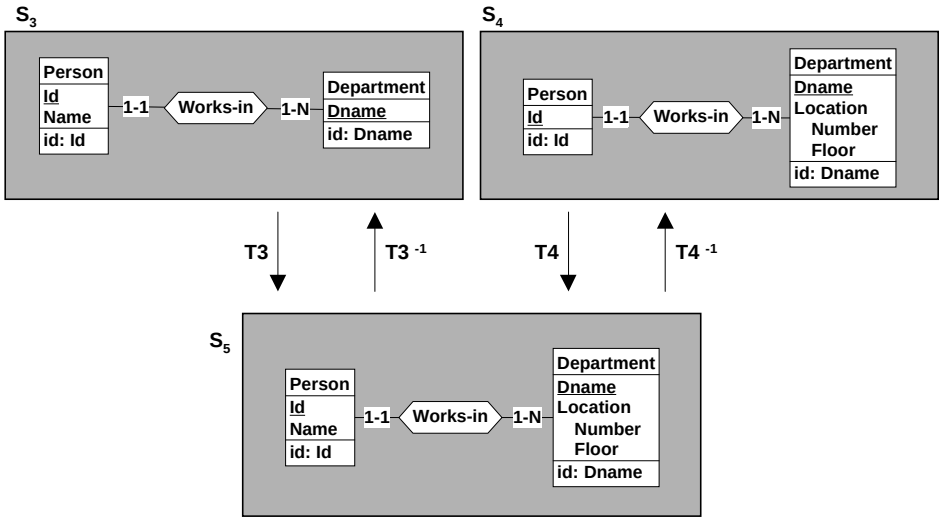


Figure 3-18: Schema integration example.

Query on S_5
[**and**, [**att**, *Person, name,* EP, Name], [**rel**, *Person, works-in, Department,* EP, DEP], [**att**, *Department, Location,* DEP, LOC], [**att**, *Location, Floor*, LOC, 4]]

Query on S_3	**Query on** S_4
[**and**, [**att**, *Person, name,* EP, Name], [**rel**, *Person, works-in, Department,* EP, DEP], [**void**], [**void**]]	[**and**, [**void**], [**rel**, *Person, works-in, Department,* EP, DEP], [**att**, *Department, Location,* DEP, LOC], [**att**, *Location, Floor*, LOC, 4]]

Query on S_5	**Query Plan**
[**and**, [**att**, *Person, name,* EP, Name], [**rel**, *Person, works-in, Department,* EP, DEP], [**att**, *Department, Location*, DEP, LOC], [**att**, *Location, Floor*, LOC, 4]]	[**and**, [**source**, S_3, [**att**, *Person, name,* EP, Name]], [**plan**, [**source**, S_3, [**rel**, *Person, works-in, Department,* EP, DEP]], [**source**, S_4, [**rel**, *Person, works-in, Department,* EP, DEP]],], [**source** S_4, [**and**, [**att**, *Department, Location,* DEP, LOC], [**att**, *Location, City, Floor*, LOC, 4]]]]

Figure 3-19: Global query decomposition and global query plan.

3.6 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.

3.6.1 History and Methodology

A history is not an arbitrary sequence of operations. It should obey to a structured way of proceeding called a methodology. A methodology specifies the products and the processes that appear when carrying out any instance of an engineering activity. This is the case for the integration process as we will see through this thesis.

Describing methodologies relates to (software/design) *process modeling* [Rolland, 1993], a discipline that is concerned with the understanding, representation and computer-based support of the software engineering activities, including the development of data structures in data-centered applications. Such activities can be modeled as a set of documents, or products (schemas, programs, specifications, etc.) and a set of engineering processes that transform input products into output products according to specific requirements to satisfy. For instance, in a classical integration process [Parent, 1998], local schemas are first translated into a canonical data model and then integrated into the unique global schema by an integration process. Each process in turn can be decomposed into a local set of products and processes, until primitive processes can be described.

The model we have developed derives from proposals such as [Potts, 1988] and [Rolland, 1993], extended to all database engineering activities. This model describes quite adequately design methodologies like integration and reverse engineering process. [Hainaut, 1996b] and [Roland, 1997] give a comprehensive specification of the model whose main concepts are the following.

- A *product* instance is any outstanding specification object that can be identified in the course of a specific design. A conceptual schema, an SQL DDL text, a COBOL program, an entity type, a table can all be considered product instances. Similar product instances are classified into products, such as the set of local schemas, entity types.
- A *process* instance is any logical unit of activity in a history which transforms a product instance into another product instance. Transforming a schema expressed in a model into an equivalent schema expressed in another model is a process instance. Similar process instances are classified into processes. Model translation is a process. There are two categories of processes, namely engineering processes and primitives. An engineering process is a goal-oriented process that is intended to make its input product satisfy specific requirements. Model translation is an example of design processes. On the contrary, a process is a primitive if it is a deterministic atomic operation. Generally, a primitive is neutral w.r.t. the requirements (it has no goal). Another difference is that the strategy of a primitive is encapsulated and is carried out by the CASE tool, while the strategy of a design process is visible, and has to be carried out by the designer, or at least under its control. The creation of an entity type and the transformation of a attribute into an entity type are examples of primitives.
- The *strategy* of a process is the specification of how its goal can be achieved, i.e. how the process must be carried out. A strategy can be deterministic, in which case it reduces to an algorithm (and can often be implemented as a primitive), or it can be non-determinis-

tic, in which case the exact way in which each of its instances will be carried out is up to the designer.

- The *hypothesis* is an essential characteristics of a process instance since it implies the way in which its strategy will be performed. When the engineer needs to try another hypothesis, (s)he can perform another instance of the same process, generating a new instance of the same product. After a while (s)he is facing a collection of instances of this product, from which (s)he wants to choose the best one (according to the requirements that have to be satisfied).
- The *history* of a process instance is the recorded trace of the way in which its strategy has been carried out, together with the product instances involved and the rationale that has been formulated. Since a project is an instance of the highest level process, its history collects all the design activities, all the product instances and all the rationales that appeared, and will appear, in the life of the project. The history of a product instance P (also called its design) is the set of all the process instances, product instances and rationales which contributed to P.

3.6.2 History Topology

The history is the trace of an actual execution of an engineering process following its strategy. As already mentioned, it can be materialized by the sequential list of operations. But, a designer can be facing a choice between several ways of performing a process. He has to make hypotheses, each one reducing the problem to a particular context, and to solve the problem in each context. All the solutions are different versions of a product. The designer can take a decision a posteriori. All the hypotheses and all solutions must be recorded in the history as well as the decisions. Each hypothesis actually starts a new sequence of operations and each decision actually brings some branches to an end. Hence, the sequence of operations must be interpreted as a *tree*.

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. This derived history is important since it describes the way the final products could have been obtained should the engineer have proceeded without any hesitation: replaying this history on the source products will yield the same output products as the actual process did.

Figure 3-20 illustrates the two representations of an history. In the left side, the history is presented as a tree made up of process instances (cn), product instances (pr) and rationales (Select). In the right side, the linear history is made up of process and product instances only.

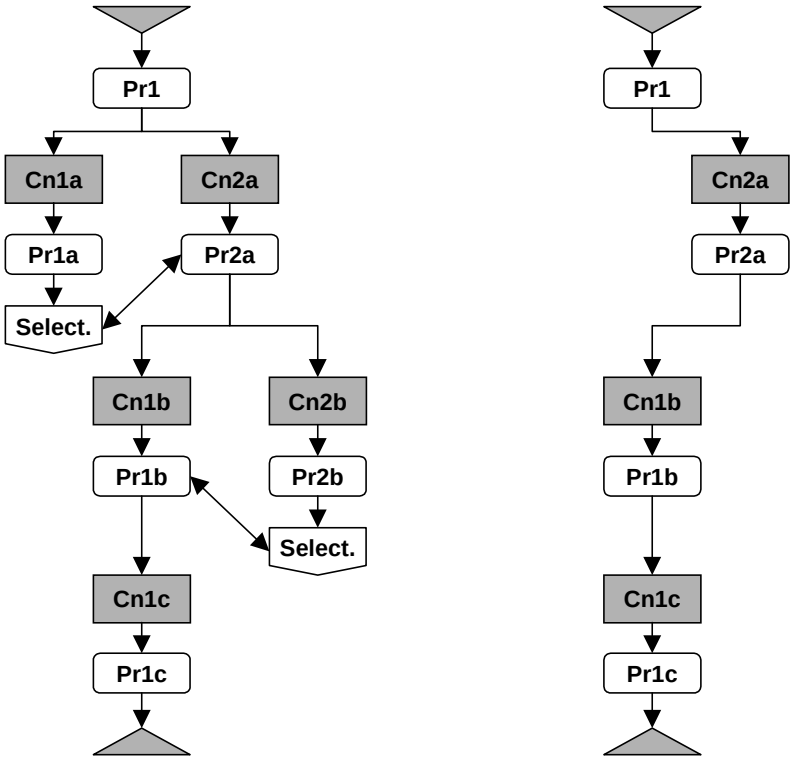


Figure 3-20: The history as a tree structure (a), and as a linear structure (b).

3.6.3 Aggregation Levels

Developing all the instances of engineering processes produces a complete history that can be given two extreme presentations:

- A *structured history* which appears as an ordered tree in which each node represents a process instance. Leaves are primitive process instances, and non-leaf nodes are engineering process instances. The immediate children of node N represent instances of the processes mentioned in the script of the process of N. The root represents the instance of the main process, i.e. the project.
- A *flat history* shows the primitive process instances only. This concept is interesting because it is the easiest form of history to record. Indeed, since it represents no engineering processes, it is methodology-neutral, and can be built by simple CASE tools. In some situations, it could be the only form of history available. Such could be the case for

loosely structured activities, such as some scenarios of reverse and integration engineering.

3.6.4 Definition

In this thesis, we consider the trace of a transformation sequence that produces schema S_j from schema S_i. This is called history (H_{ij}) of the schema transformation sequence. We note that $S_j = H_{ij}(S_i)$ where $H_{ij} = <T_1\ T_2\ T_3\ ...\ T_n>$ where T_i ($1 \leq i \leq n$) is a signature of a schema transformation. For instance, the history H_{ex} represents the schema transformation sequence of Figure 3-4.

H_{ex} = <
T_1: (R) ← FK-RT(A, {A2}, B)
T_2: (RA, A2, {A21, A22}) ← Att-ET(A,{A2})
T_3: (AA) ← Renaming(A)
>

3.6.5 Properties

History subset

A history H_p is a *subset* of history H_n (denoted by $H_p \subseteq H_n$) 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 process instances h_1, h_2, h_3, etc. We will note this decomposition $H = <h_1\ h_2\ h_3\ ...>$, where h_1, h_2, h_3 are *history slices*.

Independent histories

Let us consider history $H = <..\ T_1\ ..\ T_2\ ..>$, 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 before(T_i, T_j), that states that transformation 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:

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

Finally, T_1 and T_2 are *independent* iff:

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

Equivalent histories

Two histories (or history slices) H_i and H_j 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:

$H_0 = <h_1\ h_2\ h_3\ h_4>$,

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

If we can prove that h_2 and h_3 are independent slices, then they can be swapped in H_0, leading to history $H_1 = <h_1\ h_3\ h_2\ h_4>$. 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 3-21 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 of are expressed as foreign keys.

H : h_1: (Detail, from) ← Att-ET/Value(Order, Detail)

h_2: {OrdID} ← RT-FK(from, Detail)

h_3: {CustID} ← RT-FK(of, Account)

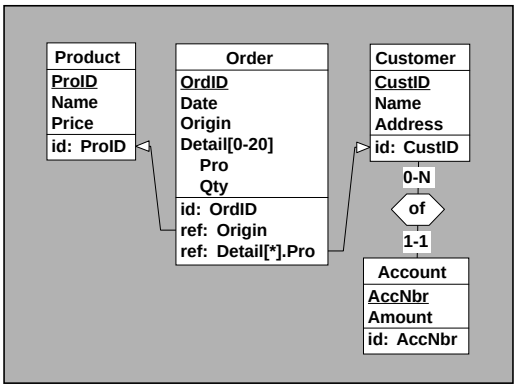


Figure 3-21: A schema example.

The graph of tr-before is as follows:

$h_1 \rightarrow h_2\ h_3$

Therefore, $<h_2\ h_3>$ are independent and swappable. The same is true of $<h_1\ h_3>$. According to the definition, $<h_3\ h_1\ h_2>$ and $<h_1\ h_3\ h_2>$ are equivalent to H, while $<h_3\ h_2\ h_1>$ is not equivalent.

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 $H' \subset H$, $H'(S) \neq H(S)$

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.

3.6.6 History Operations

History minimization

The objective is to produce a correct and minimal history as defined in Section 3.6.5. The normalization includes two processes, namely minimizing and restructuring the history:

- Removing dead branches provides a history in which only the branches and products that contribute to the final product are kept. It consists in parsing the history backward, from the final product toward the input products, and marking the process instances and product instances examined. The unmarked instances are discarded. This process is fairly easy and can be automated.
- Detecting and reducing useless sequences, and particularly useless loops are more complex problems. This problem has been formalized in [Roland, 2003]. In this thesis, we simply propose the following heuristics:

  We consider history H, which is expressed as a sequence of transformations:
  $H = \langle T_1\ T_2\ T_3\ T_4 \rangle$

  We denote by Σ_T the generic transformation of which $T \subseteq H$ is an instance. If all the following properties stand in H, then H and $\langle T_1\ T_4 \rangle$ are equivalent:

  $$\Sigma_{T2} = \Sigma_{T3}^{-1}$$
  $$C_-(h_3) = C_+(h_2)$$
  $$\neg\ (\exists\ T \in H: (T \neq T_3) \wedge C_+(T_2) \cap C(T) \neq \varnothing)$$
  $$\neg\ (\exists\ T \in H: C_+(T_3) \cap C(T) \neq \varnothing)$$

  In short, we can remove any pair of transformations which prove to be the inverse of each other, and whose target constructs are not used in any other transformations.

History inversion

Section 3.3 showed how a schema transformation can be inverted, assuming sufficient information are available in the schema transformation signature. Inverting a history is a complex task. Indeed, it implies reverting strategical decisions. If this is only possible, it is out of the scope of this thesis. So, without loss of generality because a history can be minimized, this section only deals with minimal histories.

The inverse function H_{ij}^{-1} can be derived from a function H_{ij} and can be defined as follows:

if $H_{ij} = <T_1 ... T_i ... T_n>$ then $H_{ij}^{-1} = <T_n^{-1} ... T_i^{-1} ... T_1^{-1}>$ and hence $S_i=H_{ij}^{-1}(S_j)$

In other words, H_{ij}^{-1} is obtained by replacing each origin schema transformation by its inverse and by reversing the operation order. For example, H_{ex}^{-1} is the inverted history H_{ex} representing the schema transformation sequence of Figure 3-4:

$H_{ex}^{-1} = <$
T_3^{-1}: (A) ← Renaming (AA)
T_2^{-1}: (A, {A2}) ← ET-Att(RA, A2)
T_1^{-1}: (B2) ← RT-FK(R, B)
>

This new history H_{ex}^{-1} is in fact the invert of history H_{ex}. So, in practice, inverting a history is building a new history with the reverse of the transformations of the original history inserted in reverse order.

3.7 Model Translation

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 schema transformation, query transformation and transformation history.

Model translation is defined as a *model-driven transformation* within the generic data model defined in Chapter 2. A model-driven transformation applies on a schema. It can be defined by $\mathbf{m}(M_s, M_t)$ where M_s and M_t are two different submodels, i.e., subsets of the generic data 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 action on each construct 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 relational model and the ER model (Figure 3-22). That is, a schema expressed in the relational model (M_s) is translated into an equivalent schema expressed in the conceptual model (M_t).

```
1- For each collection C, do:
        apply Del-Collection to C;
2- For each access key group AK, do:
        apply Del-AccessKey to AK;
3- For each referential group RG of an entity type ET_s that references another entity type ET_t, do:
        apply FK-RT to RG;
4- For each entity type E, do:
        If E meets the precondition of ET-RT, apply ET-RT to E;
5- For each entity type E, do:
        If E meets the precondition of ET-Att, apply ET-Att to E;
```

Figure 3-22: Sample transformation plan for translating a relational schema into a conceptual schema using the ER model.

This transformation plan can be applied to any schema expressed in the relational model. Its execution produces two result types: (1) a target schema expressed in the ER 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 relational schema (RS) of the Figure 3-23.

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

- The collection File is removed:

  T1: () ← Del-Collection(File)

- The schema includes two referential attributes (CustID of the entity type Account and CustID of the entity type Order) which express relationship types. We augment the history with the following transformations:

  T2: (R) ← FK-RT(Account, {CustID})

  T3: (passes) ← FK-RT(Order, {CustID})

- The entity type Account is an entity type attribute that is translated into a multivalued attribute of the entity type Customer:

  T4: (Customer, {Account}) ← ET-Att(Account)

- Finally, we discard all the access keys:

  T5: () ← Del-Access(Customer, {CustID})

  T6: () ← Del-Access(Order, {CustID})

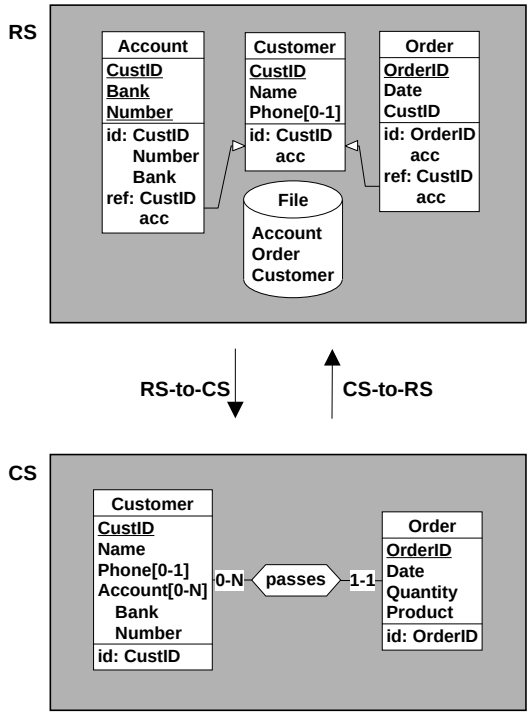


Figure 3-23: Model translation of a relational schema into a conceptual schema.

The history H of the transformation plan of the relational schema (RS) of the Figure 3-23 records the transformation sequence RS-to-CS:

RS-to-CS = <T1 T2 T3 T4 T5 T6>

```
H = <
        T1: () ← Del-Collection(File)
        T2: (R) ← FK-RT(Account, {CustID})
        T3: (passes) ← FK-RT(Order, {CustID})
        T4: (Customer, {Account}) ← ET-Att(Account)
        T5: () ← Del-Access(Customer, {CustID})
        T6: () ← Del-Access(Order, {CustID})
>
```

The conceptual schema (CS) of the Figure 3-23 is then obtained by the application of the transformation sequence LS-to-CS on LS:

CS = RS-to-CS(RS)
or CS = T6(T5(T4(T3(T2(T1(RS))))))

RS-to-CS is minimal as defined in Section 3.6.5. It can therefore be inverted by applying the inversion history operator (Section 3.6.6). The inverse history CS-to-RS is defined as follows:

CS-to-RS = $<T6^{-1}\ T5^{-1}\ T4^{-1}\ T3^{-1}\ T2^{-1}\ T1^{-1}>$
with RS = CS-to-RS(CS)
or RS = $T6^{-1}(T5^{-1}(T4^{-1}(T3^{-1}(T2^{-1}(T1^{-1}(CS))))))$

We can use the primitive transformations of CS-to-RS to automatically transform a query posed on CS into a query posed on RS by applying the substitution mechanism as defined in Section 3.5.2. For instance, Figure 3-24 shows how the request *"Find all the customers that places an order on the 1st of October 2000"* is expressed into equivalent queries in CS and RS; the equivalence being ensured by the inverse of the schema transformation T3. These queries are expressed in the query formalism. In Figure 3-25, as an example, we give the translation of the query posed on CS into an equivalent SQL query.

Query on CS	**Query on** RS
[**and**, [**att**, *Customer, CustID,* CUS, ID], [**rel**, *Customer, passes, Order,* CUS, ORD], [**att**, *Order, Date,* ORD, "01:10:2000"]]	[**and**, [**att**, *Customer, CustID,* CUS, ID], [**att**, *Customer, CustID*, CUS, CC], [**att**, *Order, CustID,* ORD, CC], [**att**, *Order, Date,* ORD, "01:10:2000"]]

Figure 3-24: Example of a query translation from CS to RS.

SQL Query on RS
Select CUS.CustID **From** Customer CUS, Order ORD **Where** CUS.CustID = ORD.CustID **And** Date = "01:10:2000";

Figure 3-25: SQL version of the query posed on RS.

Part II

Architecture

Chapter 4

Schema-oriented Architecture

In which the schema-oriented framework of data mediation is presented. The schema correspondences of the framework are formally examined by means of the transformational paradigm. This leads to the introduction of the concepts of implicit constraints and discrepancies. The role and tasks of wrappers for legacy databases are then discussed.

4.1 Introduction

This chapter presents a *schema-oriented framework* of the database mediation architecture that allows us to reduce the mediation issue in smaller independent problems. The basic idea is that we consider the schema hierarchy of the database mediation architecture instead of their implementation technologies (i.e., wrapper and mediator).

The schema-oriented framework is based on the classical schema architecture proposed by [Sheth, 1990] but differs from it in that the global schema integrates not only the legacy information but also the new requirements.

As we will see through this chapter, the schema-oriented framework of database mediation provides an elegant way to unify the architecture components and their development. Mappings defined as schema transformations are the unifying element between the architecture components and their dedicated methods. The schema correspondences of the hierarchy are formally examined by means of the transformational paradigm defined in Chapter 3.

Moreover, we discuss the important role of the *wrapper* in the particular case of a federated database architecture that integrates new requirements. We argue that some responsibilities

commonly allocated to mediators can be transferred to wrappers. To this end, we discuss a wrapper/mediator architecture where the wrapper is more than a model and query converter.

This chapter is organized as follows. Section 4.2 presents a schema-oriented framework for a database federation that integrates legacy and new components. The main goal is to propose a formal definition of schema correspondences and to present the exact contribution of each schema layer. It involves defining the concepts of *implicit constraint* and *discrepancy*. Section 4.3 presents and discusses the different possible wrapper architectures according the schema layers they emulate.

4.2 Schema-oriented Framework

In this section, we set up a *schema-oriented framework* for data mediation. The basic idea is that we consider the schema hierarchy of data mediation instead of their implementation technologies (i.e., wrapper and mediator). The schema-oriented framework provides an elegant way to unify the architecture components and their development (Figure 4-2 and Figure 4-3). Mappings defined as schema transformations are the unifying element between the architecture components and their dedicated methods: a method defines the schema transformation sequence that is used to automate the translation of queries within the corresponding architecture component. The hierarchy schemas are formally examined by means of the transformational paradigm defined in Chapter 3.

4.2.1 Definition

A *schema layer* is defined by a triple $<\{S_i\}, \{M_i\}, S>$ where $\{S_i\}$ is the set of source schemas, S is the output schema, $\{M_i\}$ is the set of mappings between each pair of $<S_i, S>$.

Mapping properties

M_i is defined as a sequence of schema transformations such as $M_i = <S_i\text{-to-}S, s_i\text{-to-}s>$ such that $S=S_i\text{-to-}S(S_i)$. M_i has an inverse: $M_i' = <S\text{-to-}S_i, s\text{-to-}s_i>$ such that $S_i=S\text{-to-}S_i(S)$. The mappings between the source and output data can be derived from the instance mappings: $s_i=s\text{-to-}s_i(s)$ where s is an instance of S and s_i is an instance of S_i.

Schema properties

We consider the function $\sigma(P)$ which gives the semantics of a product P. Two products (or schemas) can be compared based on their semantics. This refers to the *relative information capacity* of a schema [Hull, 1986]. For instance, considering the goal of the conceptual schema (CS), and adopting the conceptual formalism as a pure expression of all the semantics of the system and only it, we could write: $\sigma(CS) = CS$. However, due to the fact that several conceptual schemas can describe the same application domain (a fact sometimes called se-

mantic relativism), we will distinguish CS as a product from its semantic contents σ (CS).

If needed, we can consider that the semantics of a product is a consistent set of logic assertions such that if $P_1 \subseteq P_2$, then $\sigma(P_2) \Rightarrow \sigma(P_1)$. To simplify the discussion, we will consider that the semantics is expressed in such a way that we can also write: $\sigma(P_1) \subseteq \sigma(P_2)$. In addition this formalism must be such that $\sigma(P_1 \cup P_2) = \sigma(P_1) \cup \sigma(P_2)$.

If we consider products as schemas, we can state that the semantics between two schemas (S_1 and S_2) is preserved if these schemas can be transformed by means of semantics-preserving transformations only. In other words:

if $S_2 = S_1\text{-to-}S_2(S_1)$
such that $S_1\text{-to-}S_2$: ($\forall$ $T_i \in S_1\text{-to-}S_2$, T_i is a SR-transformation)
then $\sigma(S_1) = \sigma(S_2)$.

Seamlessly, we can also state that:

if $S_2 = S_1\text{-to-}S_2(S_1)$
such that $S_1\text{-to-}S_2$: ($\exists$ $T_i \in S_1\text{-to-}S_2$: T_i is a semantics-reducing transformation) $\wedge$
$\neg$ ($\exists$ $T_j \in S_1\text{-to-}S_2$: T_j is a semantics-augmenting transformation)
then $\sigma(S_1) \subset \sigma(S_2)$.

And, conversely,

if $S_2 = S_1\text{-to-}S_2(S_1)$
such that $S_1\text{-to-}S_2$: ($\exists$ $T_i \in S_1\text{-to-}S_2$: T_i is a semantics-augmenting transformation) $\wedge$
$\neg$ ($\exists$ $T_j \in S_1\text{-to-}S_2$: T_j is a semantics-reducing transformation)
then $\sigma(S_2) \subset \sigma(S_1)$.

4.2.2 Architecture

The architecture comprises a hierarchy of *federated layers* and *new layers* (Figure 4-2 and Figure 4-3). These layers provide a global view (NGS) that integrates the new needs in information and the existing legacy information (Figure 4-1). The methods that help developers define the schema hierarchy will be discussed separately in Chapter 6.

Hierarchy architecture

The architecture defines three classes of schemas, namely, the global schema, the federated schemas and the new schemas. The *new global schema* (NGS) meets the current global information needs by integrating the schemas of the other classes.

The *federated schemas* comprise the schemas hierarchy that describes the local existing databases. According to the general framework and according to the legacy nature of the data-

bases, each local database source is described by its own *legacy physical schema* (LPS) from which a semantically rich description called *semantically enriched legacy schema* (LP_cS), is obtained through a database reverse engineering process. LP_cS is then translated into an equivalent schema (LH_yS) expressed in a canonical data model. From this conceptual view, a subset homogenized according to the new global requirements is extracted (LH_gS); it expresses the exact contribution of this database to the global requirements. LH_gS are merged into a hierarchy of federated schemas (LFS) (a LFS can be the result of the integration of LH_gS and/or other LFS). The top of this hierarchy is made up of the federated global schema (FGS).

Finally, the *new schemas* describe the new database through its conceptual schema (NH_gS) and its physical chema (NPS). This database provides the additional required services that cannot be taken in charge by the legacy components.

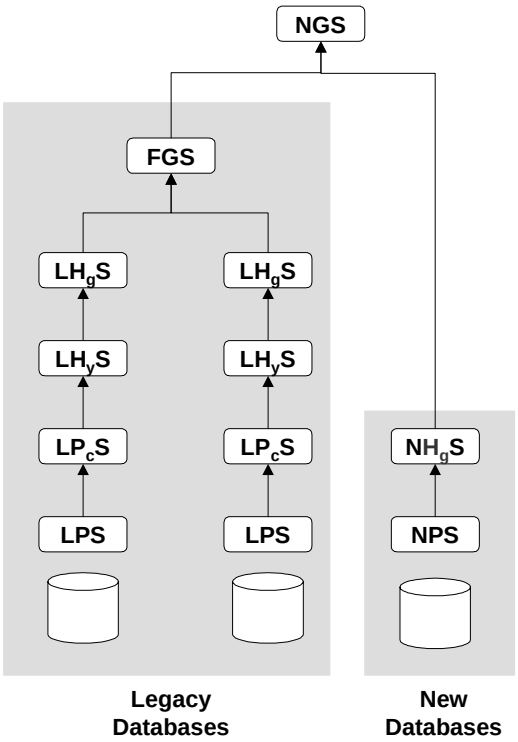


Figure 4-1: Hierarchy architecture of the schema-oriented framework. For simplicity, the federation schema hierarchy has been simplified.

Federated layers

The federated layers (Figure 4-2) provide integrated information of existing and legacy data-

bases, without the need to physically integrate them. We consider four federated layers: *semantic*, *syntactic homogenization*, *global homogenization* and *mediation*. They address four independent problems:

- *Semantic enrichment*: due to the weakness of legacy data models, the physical schema often is poor as far as semantics expression is concerned.
- *Syntactic homogenization*: each local database must deliver its data to the federation in some sort of common format, called the canonical data model.
- *Global homogenization:* when comparing a source database and the new requirements hold in NGS, different discrepancies appear, so that each source must be adapted in order to enter the federation.
- *Merging*: when the data sources have been homogenized, they can be merged (schemas and data) in order to offers the global view of all the legacy components.

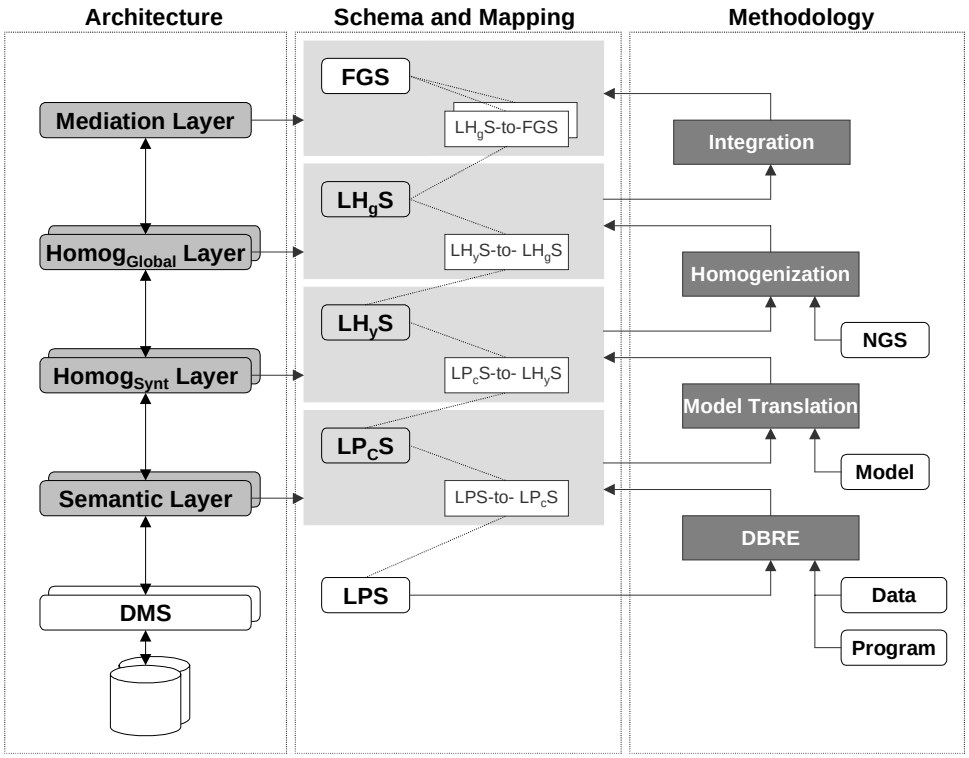


Figure 4-2: The schema-oriented framework: federation layers (semantic layer, syntactic homogenization layer, global homogenization layer and mediation layer).

New layers

The new layers (Figure 4-3) provide global information that is required by the new information system, but is not available in the legacy databases. The global mediator offers an interface based on the NGS that holds all the required information. It integrates the federated and new databases.

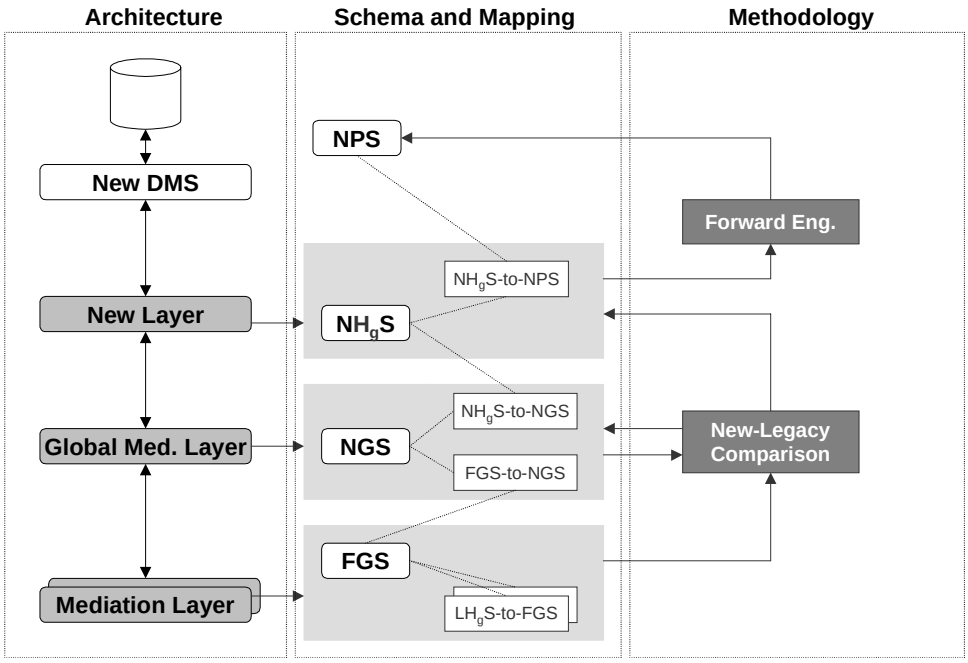


Figure 4-3: The schema-oriented framework: new layers and the federated layer of mediation.

In the following, we describe the characteristics of the federated and new layers and illustrate them by taking up the example described in Chapter 1, the common case study used throughout this thesis. We recall here the essential elements for the understanding of this chapter.

In the example of Chapter 1, we consider a company in which two manufacturing subsidiaries (M1 and M2) are active. We also consider the personnel departments P1 and P2 that ensure the HRM of each of these sites, and the sales department S, common to both. As far as this chapter is concerned, we only consider the personnel departments.

The personnel departments P1 and P2 are controlled by two independent databases, namely DB-P1 (personnel of site M1) and DB-P2 (personnel of site M2). From a technical point of view, database DB-P1 is made up of a collection of standard COBOL files, while DB-P2 was developed in Oracle V5.

Due to organizational reasons, the head office forces its branches to make their information system comply with the new business requirements that it has defined itself.

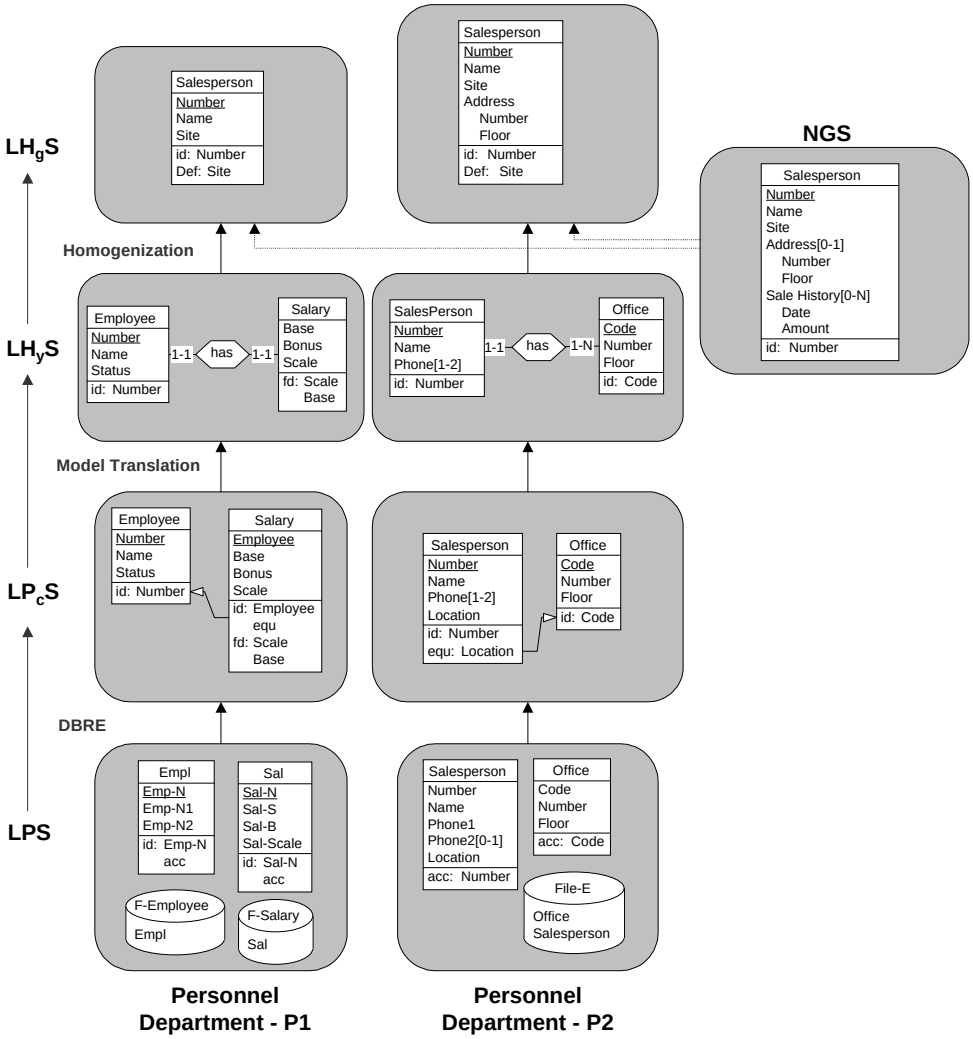


Figure 4-4: Integration example: legacy schema hierarchy. LPS/P1 comprises the record types Empl and Sal. LPS/P2 comprises two tables Salesperson and Office. In LPcS/P1, we observe the renaming of all the constructs, the elicitation of a hidden reference group and of a hidden functional dependency: Scale → Base. In LPcS/P2, a hidden foreign key, two hidden primary keys and an implicit multivalued attribute have been discovered. In LHyS/P1 and LHyS/P2, the schemas (LPcS) modeled in the generic model have been translated into schemas using the binary generic model as the canonical data model. In LHgS/P1, the entity type Employee has been homogenized accord-

ing to NGS: deletion of the attribute Status; the entity type Salary and the relationship type Has have been also discarded. In LHgS/P2, the entity types Salesperson and Office have been transformed into one complex entity type Salesperson.

4.2.3 Semantic Layer

The semantic layer is defined as follows: <{LPS}, {LPS-to-LP_cS}, LP_cS> where LPS is the legacy physical schema and LP_cS is the semantically enriched legacy physical schema. LPS is expressed in a legacy data model whereas LP_cS is expressed in the generic data model defined in Chapter 2.

Schema definition

Legacy data models cannot express all the semantics of the real world. Limitations of the modeling concepts lead to the incompleteness of the physical schema [Parent, 1998]. A federation cannot therefore assume the quality and the completeness of the physical schemas of legacy databases to be integrated.

Each legacy database D is the end product of a design process (the forward engineering FE) that started with user requirements. Whatever the way in which D was actually produced, we can imagine that it has been obtained through a chain of standard design processes that translates in a systematic and reliable way the users requirements into this database as it appears currently. Nevertheless, trying to imagine what could have happened in this hypothetical design process can be a fruitful exercise since it will allow us to identify how and why D has got to be what it currently is. In this section, we reexamine the database development process in order to identify the difference between LPS and LP_cS.

LPS definition. Roughly speaking, we can state that the physical schema (LPS) and the code (code) of an operational legacy database result from schema transformation processes:

LPS = FE(LCS)

code = COD(LPS)

where LCS is the conceptual schema, FE the forward engineering process and COD the coding process.

Ideally, we should have:

σ (code) = σ (LPS) = σ (LCS)

However, we know that empirical processes do not meet this property [Hainaut, 2002]. Let us call Δ the semantics of LCS that disappeared in the process. We get:

σ (code') $\cup$ Δ = σ (LCS)

where code' denotes the *actual* code.

We can now refine code into its DDL part and its external part:

$code' = code_{ddl}' \cup code_{ext}'$

where $code_{ddl}'$ comprises the explicit declaration of constructs, expressed in the DDL of the DMS and $code_{ext}'$ that expresses, in more or less readable and identifiable form, all the other constructs.

Accordingly, process COD comprises two subprocesses: namely, COD_{ddl} and COD_{ext}.

The discarded specifications Δ have not disappeared, but rather are hidden somewhere in the operational system or in its environment. For instance, they can be discovered in the data themselves (through data analysis techniques) or found in the exploitation and users procedures (and can be elicited by interviewing users, exploitation engineers or developers).

Now we can complete the description of the semantics properties of the physical schema and the operational code:

$code' \cup \Delta = code = FE(LCS)$

$code_{ext}' = COD_{ext}(LPS')$

$code' = code_{ddl}' \cup code_{ext}'$

with $\sigma(code') \cup \Delta = \sigma(code_{ddl}') \cup \sigma(code_{ext}') \cup \Delta$

$= \sigma(LCS)$

LP_CS definition. We observe that reversing a hierarchically decomposable process consists in inverting the order of the sub-processes, then replacing each sub-process with its inverse. For systems that have been developed according to an ideal approach, we have:

$LPS = COD_{ddl}^{-1}(code_{ddl}')$

$LP_cS = LPS \cup COD_{ext}^{-1}(code_{ext}') \cup \Delta$

Finally, we can state the schema definition of LPS and LP_cS:

- LPS is recovered by *uncoding* the actual DDL code ($code_{dll}'$);
- LP_cS is recovered by *uncoding* the actual non-DDL code ($code_{ext}'$) and *interpreting* the semantics lying outside the system (Δ); and finally merging them into LPS.

LPS only contains the structures and constraints that are *explicitly* expressed in the DDL code while LP_cS includes also the structures and constraints that are *implicitly* implemented or merely discarded during the development process. In the following section, we define the concepts of explicit and implicit constructs. We do not intend to discuss the elicitation techniques of explicit and implicit constructs here; rather, we focus on some basic developments that illustrate the structural and semantic differences between LPS and LP_cS, hence the nature

of the schema transformation sequence LPS-to-LP_cS.

Explicit and implicit constructs and constraints

As explained above, an *explicit construct* is a component or a property of a data structure that is declared through a specific DDL statement. An *implicit construct* is a component or a property that holds in the data structure, but that has not been declared explicitly. In general, the DMS is not aware of implicit constructs, though it can contribute to its management (through triggers for instance). The analysis of the DDL statements alone leaves the implicit constructs undetected. As a result, LPS holds only the explicit constraints of a database whereas LP_cS holds both its explicit and implicit constraints.

Example

The most popular example certainly is that of foreign key. Let us consider the Figure 4-5, 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 EMPLOYEE(C-ID integer primary key,
                C-DATA char 80)
create table ORDER(O-ID integer primary key,
                OWNER integer
                foreign key(OWNER) references EMPLOYEE)
```

Figure 4-5: Explicit foreign key declaration.

The program of Figure 4-6 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 EMPLOYEE(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 EMPLOYEE)
end SQL

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

Figure 4-6: Implicit foreign key.

The variety of implicit constructs can be fairly large, even in small legacy databases. In Chapter 6, we will present the most frequent implicit constructs and the transformation operators

that make them explicit.

Illustration

The physical schemas (LPS) of the personnel departments P1 and P2 are refined through DBRE in order to recover the hidden structures and constraints such as foreign keys, unique keys or multivalued attributes that are unknown in the Oracle V5 model, and referential and functional dependency groups that cannot be expressed in the standard file model. We therefore obtain the two LP_cS and the respective (*semantics-augmenting*) schema transformation sequences that model the DBRE process.

4.2.4 Syntactic Homogenization Layer

The syntactic homogenization layer is defined as follows: $<\{LP_cS\}, \{LP_cS\text{-to-}LH_yS\}, LH_yS>$ where LH_yS is the result of a *model translation*: LP_cS expressed in the generic model is translated into LH_yS expressed in the *canonical data model*.

Schema definition

The *model translation* (MOD) is a particular case of schema transformations (Chapter 3, Section 3.7). It consists in translating a schema expressed in a data model M_s into a schema expressed in another data model M_t where M_s and M_t are two different submodels (i.e., subsets) of the generic data model. Accordingly, the schema definition of LH_yS is defined as follows:

$$LH_yS = MOD_{s \rightarrow t}(LP_cS)$$

where M_s (the source model) is the generic data model and M_t (the target model) is the canonical data model.

A model translation applies *semantics-preserving transformations*[1] on the constructs of a schema expressed in M_s in such a way that the final result complies with M_t. We can therefore refine $LP_cS\text{-to-}LH_yS$:

$$LP_cS\text{-to-}LH_yS: (\forall\ T_i \in LP_cS\text{-to-}LH_yS,\ T_i \text{ is a SR-transformation})$$

Hence, we can state the semantic relation between these two schemas:

$$\sigma\ (LH_yS) = \sigma\ (LP_cS)$$

Illustration

The two schemas LP_cS of personnel departments are translated into the binary generic data model described in Chapter 2. This model includes the concept of one-to-many association,

1. The fact that a model translation can include transformations that remove technical constructs like access keys or process units is irrelevant here.

while a LP_cS schema makes use of foreign keys. The current layer will express each foreign key as an association type.

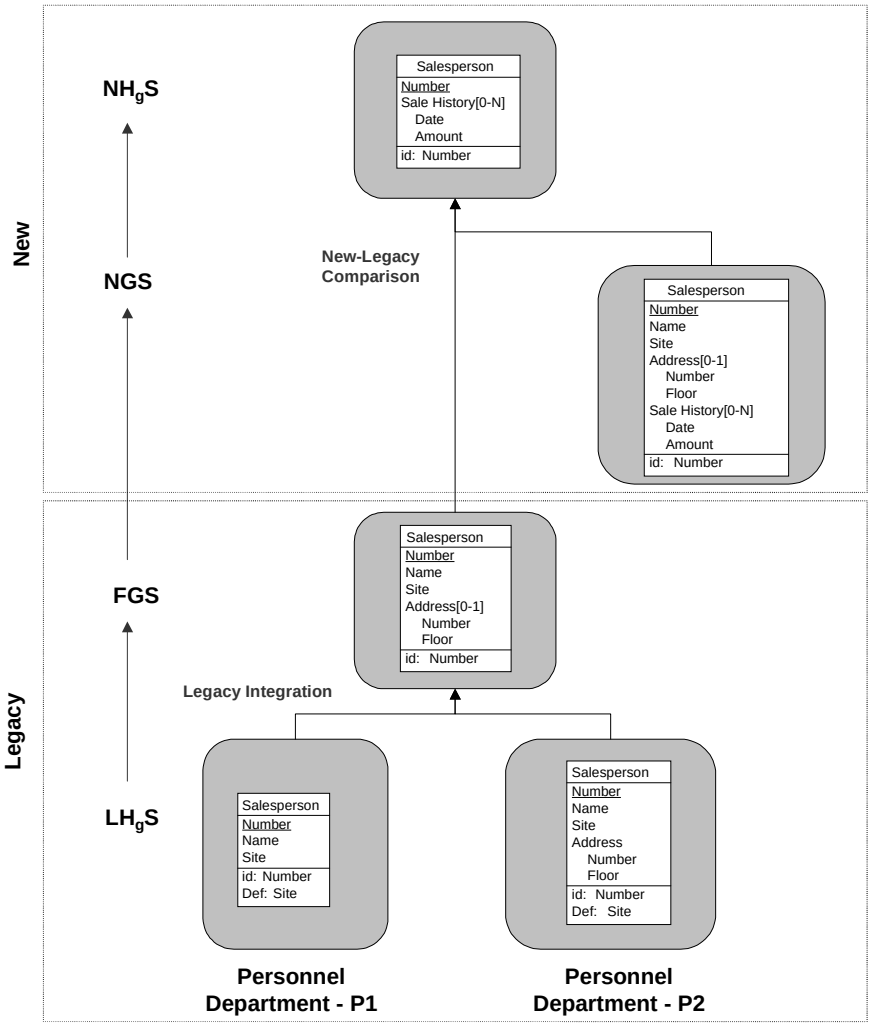


Figure 4-7: Integration example: legacy an new schema hierarchy. LHgS/P1 and LHgS/P2 are integrated into a common entity type Salesperson (FGS). By comparing the NGS against FGS, we define NH_gS that holds all the new constructs and an attribute (Number) common to the FGS.

4.2.5 Global Homogenization Layer

The global homogenization layer is defined as follows: <{LH_yS}, {LH_yS-to-LH_gS}, LH_gS> where LH_gS is the legacy schema homogenized according to the new global schema (NGS). Global homogenization (HOM) removes some *discrepancies* of legacy databases in regard to NGS [Yan, 1997]. These discrepancies arise when legacy databases model the same application domain differently.

Schema definition

Roughly speaking, LH_gS can be defined as the intersection of NGS and LH_yS for which all the discrepancies are removed:

$$LH_gS = HOM_{NGS}(LH_yS) \cap NGS$$

Since the discrepancies are removed by applying semantics-preserving or semantics-reducing transformations on LH_yS, we can refine the schema transformation sequence LH_yS-to-LH_gS:

$$LH_yS\text{-to-}LH_gS: \neg (\exists\ T_i \in LH_yS\text{-to-}LH_gS: T_i \text{ is a semantics-augmenting transformation})$$

Hence, we can state the following semantics relationships between LH_gS and LH_yS; between LH_gS and NGS:

$$\sigma (LH_gS) \subseteq \sigma (LH_yS)$$

$$\sigma (LH_gS) \subseteq \sigma (NH_gS)$$

Discrepancy

The data and schema expressed in LH_yS can exhibit discrepancies against NGS:

- *Domain discrepancies*: different units of measurement (Euro vs. Dollar or legacy currencies); different notation (“firstname lastname” vs. “lastname, firstname”); different representation (ISBN with dashes vs. with spaces);
- *Syntactic discrepancies*: besides the usual conflicts related to synonyms and homonyms, a syntactic conflict occurs when the same concept is presented by different constructs in local schemas. For instance, an same concept can be represented by an entity, by an attribute value and by a relationship.
- *Semantic discrepancies*: 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.

All these discrepancies can be resolved by means of schema transformations. In Chapter 6, we will present the most frequent discrepancies and the transformation operators that solve them.

Illustration

Figure 4-4 depicts some discrepancies between LH_yS of the two sources and NGS: (1) *domain discrepancy:* in LH_yS, salesperson numbers are recorded with dash separation while the same data are to be without any dash in NGS; (2) *syntactic discrepancy:* Address and Office are synonyms; (3) *semantic discrepancy*: Office is modeled as an entity type in LH_yS of the site P2 while it is modeled as a compound attribute in NGS; NGS only considers employees that work as seller whereas the local database records all the employees that work in the site P1 whatever their status (the value domain of Status of Employee is {Salesperson, Secretary, Manufacturer, Manager}.

4.2.6 Mediation Layer

The mediation layer is defined as follows: $<\{LH_gS_1,..., LH_gS_n\}, \{LH_gS_1\text{-to-FGS},...,LH_gS_n\text{-to-FGS}\}, FGS>$. The FGS is defined as the union of all the legacy databases through their LH_gS:

$$FGS = \cup_i(LH_gS_i)$$

Each FGS obtained as the result of a schema transformation must satisfy certain quality criteria. The notion of schema transformation sequence can be used to express such criteria. [Batini, 1986] introduced four criteria for global schema: namely, *correctness*, *completeness*, *minimality* and *understandability*. In the following, we consider these quality criteria with respect to the notion of schema transformation sequences between each LH_gS_i and FGS:

- *Correctness.* FGS must be able to correctly represent all the concepts of LH_gS_i. In fact, the global schema must be derivable from local schemas by correct schema transformations. In this context, correctness refers to the fact that the semantics of FGS should not be greater than the semantics of the union of LH_gS_i: $\sigma(FGS) \subseteq \sigma(\cup_i(LH_gS_i))$.
- *Completeness.* FGS must be able to represent all information representable by the union of the LH_gS_i. Thus, the semantics of the global schema must be greater than or equal to the information capacity of the union of LH_gS_i: $\sigma(\cup_i(LH_gS_i)) \subseteq \sigma(FGS)$.
- *Minimality.* Same real-world concepts must be represented in FGS only once. The objective of the minimality criterion is to discover and eliminate redundancies at the schema level. Redundancy elimination corresponds to a semantics-preserving transformation sequence.
- *Understandability.* The objective of the understandability criterion is to choose the most understandable representation. FGS *meets* this objective since it integrates schemas (LH_gS_i) homogenized according to the new requirements.

Correctness and completeness together require a semantics-preserving transformation sequence between each FGS and LH_gS_i ($\sigma(FGS) = \sigma(\cup_i(LH_gS_i))$).

Illustration

Figure 4-7 depicts the integration of the two homogenized schemas of the two personnel departments. Since the instance value sets of the two databases are distinct, the integration process is immediate.

4.2.7 New and Global Mediation Layers

The new layer is defined as follows: <{NGS, FGS}, {NGS-to-NH_gS}, NH_gS> where NH_gS is the new local schema homogenized according to the new global schema (NGS). NH_gS provides the information required by the new system (NGS) but not supplied by the legacy databases (FGS). Since the new layer has to operate in collaboration with the federated layers, NH_gS must include information common to FGS. For instance, some of the important entity types that appear in FGS can also appear in NH_gS, either as duplicates (e.g., to represent additional attributes of an entity type, or to improve performances), or as extension (e.g., the personnel of a third department has to be incorporated) of the former ones.

Therefore, NH_gS can be defined as:

$$NH_gS = (NGS \setminus FGS) \cup COR(NH_gS, FGS)$$

where COR is the correspondence detection process.

The main goal of the correspondence detection process is the identification of overlapping or complementary information between NH_gS and FGS. The goal of this process is obvious:

$$NH_gS \cap FGS \neq \varnothing$$

Finally, we can state the semantics definition of NH_gS:

$$\sigma(NH_gS) \supseteq \sigma(NGS \setminus FGS)$$

Illustration

By comparing the NGS against FGS, we define NH_gS that holds all the new constructs (sale history) and a construct (Salesperson.Number) common to the FGS (Figure 4-7). Here, the connection with the legacy databases is made possible through a common identifier.

4.2.8 Synthesis

We are now able to summarize the schema and mapping properties of the schema-oriented framework. As a significant result, please note the central role of the semantic and homogenization layers in the hierarchy: they support the new layers as they reduce the domain, syntactic, semantic and instance distances between the NGS and each source.

Schema properties

Figure 4-8 depicts the schema definition and the semantics they hold. It summarizes the main aspects of the database federation we take into account in this thesis:

- *Legacy aspects:* LP_cS holds not only the information explicitly declared in the DDL code but also the extra semantics that is hidden elsewhere.
- *New aspects:* LH_gS, NH_gS and NGS integrate the new needs of the organization in the database federation system.

Schema	Schema Definition	Semantic Definition
LPS	$LPS = COD_{ddl}^{-1}(code_{ddl}')$	$\sigma(LP_cS) = \sigma(code_{ddl}')$
LP_cS	$LP_cS = LPS \cup COD_{ext}^{-1}(code_{ext}') \cup \Delta$	$\sigma(LP_cS) = \sigma(LPS) \cup \sigma(code_{ext}') \cup \sigma(\Delta)$
LH_yS	$LH_yS = MOD(LP_cS)$	$\sigma(LH_yS) = \sigma(LP_cS)$
LH_gS	$LH_gS = HOM(LH_yS) \cap NH_gS$	$\sigma(LH_gS) \subseteq \sigma(LH_yS)$
FGS	$FGS = \cup_i(LH_gS_i)$	$\sigma(FGS) = \cup_i(\sigma(LH_gS)_i)$
NH_gS	$NH_gS=(NGS\setminus FGS) \cup COR(NH_gS, FGS)$	$\sigma(NH_gS) \supseteq \sigma(NGS \setminus FGS)$

Figure 4-8: Schema definition synthesis.

Mapping properties

From the schema transformation sequence definitions described above, we are now able to state the properties of the schema layer mappings by using the history properties defined in Chapter 3, Section 3.6:

- *For a specific schema layer:* the mappings are defined as a minimal sequence of schema transformations;
- *For the sequence of the schema layers:* the transformation sequences are not equivalent but are dependent. We can observe an order relation between them: before(LPS-to-LP_cS, LP_cS-to-LH_yS); before(LP_cS-to-LH_yS, LH_yS-to-LH_gS); before(LH_yS-to-LH_gS, LH_gS-to-FGS).

Note that a sequence comprising the transformations of two (or more) minimal dependent sequences is not necessary minimal (Chapter 3, Section 3.6). Let us consider a transformation sequence LPS-to-LH_gS defined on the semantic layer, syntactic and semantic homogenization layers:

$$LPS\text{-to-}LH_gS = \langle LPS\text{-to-}LP_cS \quad LP_cS\text{-to-}LH_yS \quad LH_yS\text{-to-}LH_gS \rangle$$

Although the subsequences LPS-to-LP_cS, LP_cS-to-LH_yS and LH_yS-to-LH_gS are defined as minimal and dependent, the sequence LPS-to-LH_gS is not necessary minimal since it can in-

clude a transformation in a subsequence that has its inverse in another one or a transformation on an construct that do not contribute to the final schema LH_gS (Chapter 3, Section 3.6.5). We must therefore apply the minimal operator on the sequence LPS-to-LH_yS:

$$LPS\text{-to-}LH_gS = \min(<LPS\text{-to-}LP_cS \quad LP_cS\text{-to-}LH_yS \quad LH_yS\text{-to-}LH_gS>)$$

Example

In Figure 4-7, LH_gS of the personnel department P1 is the result of three schema transformation sequences between the schemas LPS, LP_cS, LH_yS and LH_gS. In these sequences, all the transformation schemas applied on the entity type Sal(ary) can be discarded since this construct doesn't belong to LH_gS.

4.3 Wrapper Architecture

The schema-oriented framework is used to examine the role of a wrapper in the particular case of a federated database architecture that integrates new requirements. We argue that some responsibilities commonly allocated to mediators can be transferred to wrappers.

4.3.1 Wrapper and Schema Layers

Whatever the underlying technologies proposed in the literature, we can hypothesize that their wrappers integrate at least the syntactic homogenization layer. Figure 4-9 represents the wrapper architectures in which other schema layers are integrated or not into the wrapper.

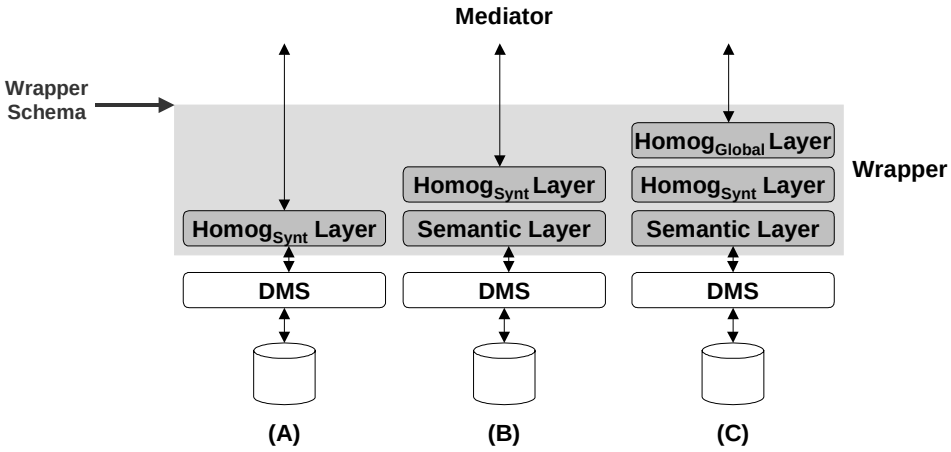


Figure 4-9: Wrapper and schema layers: (a) the thin wrapper approach; (b) the conceptual wrapper approach and (c) the thick wrapper approach.

The architecture depicted in Figure 4-9 (a) shows the most classical wrapper architecture (called the *thin wrapper approach*) where the wrapper only includes the syntactic layer whereas all the mediation tasks are assigned to the mediator (e.g., [Vermeer, 1996], [Hammer, 1997], [Bouguettaya, 1998], [McBrien, 1999]). Such wrappers are based on the quality and completeness of the database structures to be wrapped that cannot be relied on in many practical situations. They cannot prevent inconsistencies caused by a violation of an implicit constraint since they do not emulate such concepts.

Figure 4-9 (b) represents the *conceptual wrapper* that offers a schema that contains and incorporates extra semantics (implicit constraints and constructs) that are not defined in the physical schema of the underlying database. This architecture, implemented in the InterDB Project [Thiran, 2001], aims at providing a wrapper interface that allows local consistent updates of the legacy data. Note also that the wrapper is only defined from information of a local and legacy database source. That is, it does not take in charge any information from the other sources nor from the global schema.

Figure 4-9 (c) represents the *thick wrapper architecture* in which the wrapper offers not only the conceptual interface of the underlying legacy database but also an interface homogenized with regard to NGS. As a result, the thick wrapper supports the mediator as it reduces the domain, syntactic, semantic distances between the NGS and each source. However usual problems associated with view updates need to be addressed [Dayal, 1981]. This thick wrapper approach especially fits very well situations where new local applications have to integrate the new functions and requirements of the organization.

In essence, these three approaches of database federation architecture provide the bases for a transition strategy from mediator-centered to wrapper-centered architectures in which the wrapper includes not only the syntactic layer but also the semantic and homogenization layers that are traditionally allocated to the mediator. Figure 4-10 summarizes the main characteristics of a wrapper for the three approaches.

Criteria / Approach	Thin Wrapper	Conceptual Wrapper	Thick Wrapper
Schema Layers	Syntactic Homog.	Semantic Syntactic Homog.	Semantic Synactic Homog. Semantic Homog.
Wrapper schema	LPS	LP_cS	LH_gS
Wrapper mapping	Min(LPS-to-LH_yS)	Min(<LPS-to-LPcS LP_cS-to-LH_yS>)	Min(<LPS-to-LP_cS LP_cS-to-LH_yS LH_yS-to-LH_gS>)
Mapping nature	Semantics-preserving	Semantics-preserving ∨ Semantics-augmenting	Semantics-preserving ∨ Semantics-reducing ∨ Semantics-augmenting
Update	Inconsistent	Consistent	Consistent but same problems than SQL views

Figure 4-10: Main characteristics of wrappers.

4.3.2 Wrapper Architecture Discussion

Defining the layer location in a wrapper/mediation architecture is an important issue that must be tackled. More specifically, the question is:

> *Where should the semantic and/or the global homogenization layers be located? Have they to be managed by the wrapper?*

Though no complete answer can be given to this question, we can nevertheless identify six major dimensions and criteria for the layer location: *methodological, organizational, optimization, reuse, local applicability and salability aspects.*

Methodological aspects

Figure 4-11 shows the methodology pattern for developing architecture components. The methodology template shows the two main stages, namely the wrapper building and the mediator building. Wrapper stages can be done independently. The portion of effort spent in each activity is represented by the surface covered by each polygon. All the polygons form a triangle that represents the global effort in building a database federation. An horizontal line separates the mediator stage from all the wrapper stages. The horizontal line position depends on the service layers supplied by the mediator or the wrappers. For instance, if the wrapper only includes the syntactic homogenization layer, the horizontal line position is low, i.e., the wrapper stage surface is small.

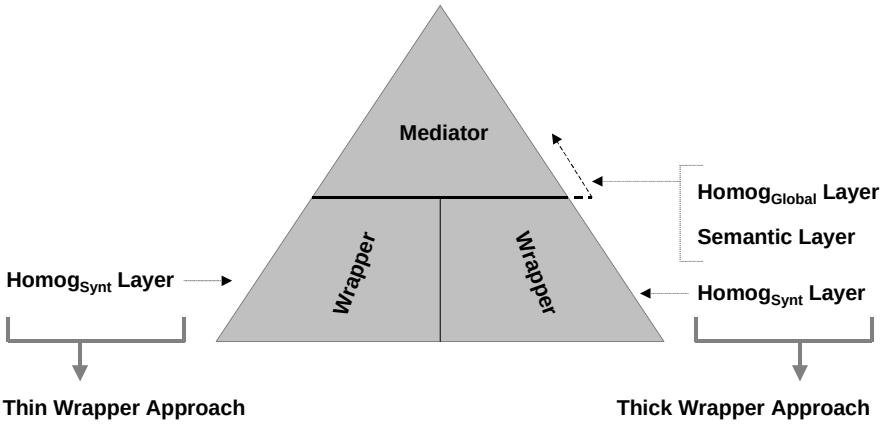


Figure 4-11: Wrapper and mediator stages and their horizontal border.

As an illustration, Figure 4-11 points to the main assets for the thick wrapper approach:

- *Top position of the horizontal line:* the wrapper stage involves only one legacy data source (and the NGS). Once conceptualized and homogenized, a database can be integrated into NGS through a simple integration mechanism. The thick wrapper methodology is heavy in semantics recovery and homogenization but light in integration. The semantics recovery and homogenization count for the most of the effort for including a new database in the federation (Chapter 6).
- *Maximization of development distribution:* wrapper stages can be done in parallel; these stages can be independently done by analysts having a strong knowledge of the databases.

Organizational aspects

The thick wrapper approach requires more responsibility from the local database administrator. Though this can induce negotiation problems, the local administrator could feel more involved in building the federation (Chapter 6, Section 6.5.1). In addition, the problems are solved where they are best mastered (Chapter 6, Section 6.3.2).

Optimization aspects

The execution cost of a query received by a mediator is expressed as a weighted combination of I/O, CPU, and communication costs. A typical simplification [Öszu, 1999] is to ignore local processing cost by assuming that the communication cost is dominant. Important inputs to the optimizer for estimating execution costs are fragment statistics and formulas for estimating the cardinalities of results. A common formula for determining the total time (CT) is: $CT = T_{MSG} + (T_{TR} * \#bytes)$. T_{MSG} is the fixed time of initiating and receiving a message,

while T_{TR} is the time it takes to transmit a data unit from one site to another; #bytes is the sum of the sizes of all messages.

The homogenization layer can apply operations of data selection and projection (Chapter 6, Section 6.5.2). Hence, if the homogenization layer is integrated into the wrapper, the cost of transferring data over the network (#bytes) can be reduced.

Reuse aspects

Service layers manage the same kind of mappings, so that integrating them into a same architecture component can maximize some program reuse. This concerns the location of the semantic and homogenization layers. Indeed these layers manage mappings of the same nature (for instance, *naming mapping, domain mapping*). Such mappings could be simplified and reused if they are managed by the same architecture component.

In Section 4.2.8, we noted that a transformation sequence which integrates two (or more) minimal (no transformation can be removed) independent transformation sequences is not necessary minimal. Therefore, merging the sequences and performing their actions on the same site can lead to avoid unnecessary mapping and hence can lead to better performance.

Local applicability aspects

The semantic layer offers an enriched schema that holds all the explicit and implicit structures and constraints of a legacy database, and therefore offer added value from the semantic point of view. Hence, it allows new local applications to perform consistent updates of the legacy data.

The primary goal of the homogenization layer is to present an homogenized view of the legacy databases for the mediation layer. However, the homogenization layer can also be used to enhance and evolve local applications since it allows them to access the legacy databases through the new interface. For example, the homogenization layer can introduce new data formats or domains that can be directly used by the new local applications. As a consequence, if new local applications must integrate the new requirements of the organization, the thick wrapper approach should prove better than the thin wrapper approach. This feature plays an essential role when the legacy system is intended to be gradually migrated into a newer and modern system [Brodie, 1995]. During the migration stage, legacy and new systems must coexist together [Umar, 1997]. Hence, the importance of using the thick wrapper approach that wraps the legacy databases in such a way that the new applications can be developed on top of it.

Example

The homogenization layer translates legacy currencies, such as Belgian Francs in Euros. New applications can now extract data from the local database without worrying about obsolete currencies.

Scalability aspects

The thick wrapper approach is designed to facilitate dynamic and scalable data integration. Its complexity is independent of the number of sources registered. Thick wrapper methodology is heavy in semantics recovery and homogenization but light in integration. Semantics recovery and homogenization which can be done independently and in parallel, account for most of the effort for including a new database in the federation. Including a new database in the federation involves modification that mainly is taken in charge by the local maintainers and owners (Chapter 6).

Chapter 5

Wrapper Architecture

In which the technology of wrappers for legacy databases is presented. The query processing including the semantic implicit integrity control is described. Our experience in building wrappers is then presented and some issues of wrapper generation are addressed. The architectures of operational wrappers for relational and standard file databases - the InterDB wrappers - are finally presented.

5.1 Introduction

As we already mentioned in Chapter 1, legacy data systems contain information that is embedded in legacy databases 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 (Chapter 4).

A *wrapper* attempts to extend the usefulness of the legacy data systems by facilitating their integration into modern (distributed) systems. In general, data wrapping is a very attractive strategy due to several reasons. First, it does not modify the legacy data systems. Secondly, it addresses the challenge of database heterogeneity by providing a standard and common interface. This interface is made up of: (1) a *wrapper schema* of the wrapped database, expressed in a canonical data model and (2) a *common query language* which uses the semantics defined in the wrapper schema. Queries on the wrapper schema are also known as *wrapper queries*.

Here follow some of the most frequent applications of database wrapping:

- *Legacy system integration.* As already mentioned in the previous chapters, the wrapper is one of the architecture components of a database federation.
- *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 data warehouses 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 opening to modern system.* Legacy systems provide services 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 extension*. This term designates changing and augmenting the functional goals of a legacy system, such as adding new functions, or its external behavior, such as improving its robustness or its security.

The wrapper technology that we will describe through this chapter has been tested and applied within the framework of the different applications described below. In Figure 5-1, we give synopses of the main case studies to illustrate the key applications of our wrappers.

Wrapper Application	References	Comments
Data integration	[Thiran, 1998] [Hainaut, 1999]	Our wrapper is used in a federation architecture
Data migration	[Henrard, 2002]	A variant of our wrapper is used within the framework of an incremental application migration
Data extraction	[Delcroix, 2001]	By accessing legacy data through our wrapper, a data converter can be used whatever the underlying DMS

System extension	[Thiran, 1999]	An implementation of our wrapper offers a Java remote interface of Cobol files

Figure 5-1: Current applications of our wrappers.

5.1.1 Wrapper Baselines

As we have pointed out, wrappers are crucial because they are the focal point for managing the diversity of legacy databases [Roth, 1997]. Below a wrapper, each data source, or data management system, has its own data model, schema, programming interface, and query capability. The data model can be relational, object-oriented, or specialized for a particular domain. 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., sequential files of records). Somewhat more sophisticated data systems allow a record ordering to be specified, and apply elementary predicates to limit the amount of data retrieved. At the other end of the spectrum are databases like relational databases that support complex predicates and operations like joins or aggregations.

Several prototype wrapper systems for databases have been developed. They share two common characteristics according to which they can be compared, namely the query-oriented definition of their mappings and the level of transparency they provide.

Mapping definition

One of the most challenging issues in wrapper systems is the definition of the mappings. Two main basic approaches have been used to specify them.

The first and very widespread approach ([Vermeer, 1996], [Garcia, 1997]) is *query-oriented* in that it provides mechanisms by which users define wrapper schema constructs as view over source schema constructs, but do not focus on the semantics of the data sources. More recent approaches on automatic wrapper generation ([Vidal, 1998], [Roth, 1997], [Hammer, 1997]) are also query-oriented.

In contrast, the second approach ([Thiran, 1998], [McBrien, 1998]) is *schema-oriented* in that mappings are defined as schema transformations that are used to automate the translation of queries. In Chapter 3, we showed how schema transformations can be used as a sound basis for the query translation.

Wrapper transparency

Several authors ([Vermeer, 1996], [Hammer, 1997], [Bouguettaya, 1998], [Roth, 1997]) consider *a wrapper as a data model converter*, i.e., a software component that translates data and queries from one data model, generally the legacy DMS model, to another, abstract, DMS-independent, model. That is, the wrapper is only used to overcome the data model and query

heterogeneity in database systems, leaving aside the semantic aspects of wrapping.

Moreover, schema translation in the above is on a model by model basis: the translation of schemas and queries of a specific data model to schemas and queries of another specific data model. Examples of such wrappers are given in Figure 5-2. No general techniques that allow the translation of a schema described using a given model into an equivalent schema described using another model are provided. This issue is addressed in [Thiran, 1988] and [McBrien, 1999]. We both propose an extensible framework that allows the description of arbitrary models and the use of schema transformations between two no necessary equivalent schemas. We differ from the approach of [McBrien, 1999] in using a high-level generic data model that allows to easily represent any constructs and constraints whatever their underlying data model and their abstraction level (Chapter 2).

Wrapper Model	Legacy Model	References
Relational	Hierarchical or network	[Gray, 1984] [Rosenthal, 1985]
Entity-Relationship	Relational	[Markowitz, 1993]
Object-oriented	Relational	[Kim, 1990] [Meng, 1995] [Bergamaschi, 1997] [Vermeer, 1996]
XML	Relational	[Gardarin, 1999] [Manolescu, 2001b] [Rodríguez, 2001]

Figure 5-2: Some research on wrappers dedicated to particular model and query languages.

Finally, in terms of query language, the transparency is not necessary guaranteed by the current wrapper prototypes. Their query language power can indeed vary according to the query capability of the underlying DMS. This requires a wrapper writer to describe legacy DMS (and wrapper) capabilities by means of a query capability declarative specification language. Examples of such systems are Tsimmis [Papakonstantinou, 1995], Disco [Kapitskaia, 1997] and Information Manifold [Levy, 1996].

The idea of declarative specifications of query power is attractive, but there are some problems with this approach [Roth, 1997]. One of them is the fact that the resolution of complex query mapping is not solved at the wrapper level whereas choosing legacy query plan can require detailed knowledge of the contents, semantics, and physical organization of the legacy data. Another problem is that capabilities and restrictions are difficult or impossible to express declaratively. The limits vary for different DMS and even for different versions of the same DMS.

5.1.2 Proposals

In this thesis, we develop an *updatable*[1] wrapper architecture for legacy databases. First of all, we consider the DMS heterogeneity among the legacy databases. Our goal is to provide a wrapper architecture that guarantees both the *query and model transparency*. A key characteristic to our approach is that the new applications that access to legacy data through the wrapper do not need to track the minute details of the capabilities and restrictions of the underlying DMS. Instead, the wrapper encapsulates this knowledge and ensures that the query plans that it produces can actually be executed by the underlying DMS.

Moreover, motivated by the legacy nature of existing databases, we address the *semantic aspects of the wrapping*. Our focus lies on the use of the reverse-engineering for defining the wrapper schema and the use of schema transformations for defining query mapping. As a result, our wrappers do not only provide the DMS transparency but also offer a semantically rich description of the underlying databases.

Finally, we investigate the *semi-automated development of wrappers*. Our approach of wrapper development addresses the challenge of wrapper generation by using the schema transformation approach described in Chapter 3.

> *Here, our idea is to semi-automatically develop an updatable wrapper technology for legacy databases that guaranties both the DMS and local semantic transparencies.*

5.1.3 Chapter Organization

The chapter is organized as follows. Based on our experience in legacy data systems, Section 5.2 explores the features of wrappers for legacy data systems. Section 5.3 presents the architecture of wrappers according to three views: system-oriented, schema-oriented and query-oriented. Section 5.4 develops the query processing. This relies on the schema transformation approach and the query capabilities of legacy DMS. Section 5.5 deals with developing wrappers in a semi-automated way. Hardcoded wrappers are generated from schema transformations and specifications. Section 5.6 presents two operational wrappers developed as part of the InterDB Project [Thiran, 1998]: wrappers for COBOL files and wrappers for relational databases [Thiran, 1999].

1. An updatable wrapper is a wrapper that allows update queries.

5.2 Wrappers and Legacy Databases

5.2.1 Definition and Main Features

Roughly speaking, 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).

The functionality of a data wrapper can be subdivided into a number of categories. In the following, we present the main classes of functionality that are relevant for our work:

- *Query and data translation.* 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 client application.
- *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.
- *Semantic integrity control*. Wrappers emulate (implicit) integrity constraints defined at their interface but not declared in the underlying database definition.
- *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 client application.
- *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.
- *Concurrence and recovery control.* This function is to permit concurrent updates of the underlying legacy databases. This includes transaction and failure management.

5.2.2 Legacy Issues

Wrapping legacy data systems poses complex problems. In this section, we discuss some important specificities of legacy systems and their influence over the wrapper development. The specificities we propose to analyze are: (1) their autonomy, (2) their data model, (3) their physical schema, (4) their access language. The main objective of this section is to draw up a list of requirements of wrappers which are intended to encapsulate legacy data systems.

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 application access without modify them. Thus, it is important to understand the aspects of autonomy and their influence over the wrapper development (Figure 5-3).

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 [Garcia, 2000] can be not available for the wrapper. Hence, the cost model of the legacy DMS can be unavailable or cannot be calibrated [Lim, 1999].

Moreover, the legacy data systems can be subject to change and evolution. Indeed, these autonomous systems can be modified in the context of local requirements. Frequent modifications of the legacy data system can preclude wrapper viability. However, the fact that the data systems are legacy means that they significantly resist modification and evolution [Brodie, 1995]. Without *too much* reducing the wrapper applicability, we can therefore hypothesize that the legacy data systems are stable.

Legacy application autonomy. Legacy application autonomy means that the wrapper cannot influence the way the individual legacy applications access data. It means that the wrapper can not interfere with their execution. A main problem is to preserve the *legacy data consistency*. As already stated in Chapter 4, the legacy DMS only manages the explicit constraints whereas the implicit constraints are implemented in the local application programs. In order to preserve the legacy data consistency, both constraints must be considered. As a result, the wrapper must guarantee legacy data consistency by rejecting updates that violate implicit constraints.

Legacy Feature	Wrapper Requirement
DMS autonomy	Wrapper interacts with the underlying database through its LPS
DMS autonomy	Wrapper uses the query language of the legacy DMS (Section 5.4)
Legacy Application autonomy	Wrapper must guarantee legacy data consistency as it is specified in the whole legacy system (Section 5.4.6)

Figure 5-3: Legacy data system autonomy and wrapper requirement.

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 canonical (common) data model. As already mentioned in the previous chapters, this model transformation can be modeled as schema

transformations. These transformations have to consider the modeling power of the data models. If the modeling power of the canonical data model is weaker than that of the legacy data models, it can happen that some model-inherent constraint have to be made implicit. In the opposite direction, if the canonical data model has more modeling power than a legacy data model, it is possible that certain implicit constraints can be transformed into explicit constraints. As a result, it is usually expected that the modeling power of the canonical data model is richer than that of the legacy data models [Sheth, 1990] (Figure 5-4).

Legacy Feature	Wrapper Requirement
Data model heterogeneity	Wrapper includes the syntactic homogenization layer (Chapter 4, Section 4.2.4)
Data model expressiveness	The canonical data model must be at least as expressive as any legacy data model (Chapter 2)

Figure 5-4: Legacy data model heterogeneity and wrapper requirement.

Legacy physical schemas

As demonstrated in Chapter 4, a wrapper cannot 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 legacy data wrapping and *reverse database engineering* [Thiran, 2001]. This will be discussed separately in Chapter 6.

Legacy Feature	Wrapper Requirement
Physical schema incompleteness	Wrapper includes the semantic layer (Chapter 4, Section 4.2.3)
Physical schema incompleteness	Wrapper development requires DBRE (Chapter 6, Section 6.4)

Figure 5-5: Legacy physical schema and wrapper requirement.

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).This prevents a uniform treatment of queries managed by wrappers. 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.

Legacy Feature	Wrapper Requirement
Query language	Wrapper must be aware of legacy DMS query capabilities (Section 5.4)
Query language heterogeneity	Wrapper development depends on the DMS query capabilities (Section 5.5)

Figure 5-6: Legacy query language and wrapper requirement.

Example

To illustrate this point, let us consider an application that issues a query to a wrapper. The underlying legacy data it accesses are recorded in COBOL files. For simplicity, we assume that the wrapper query language is SQL.

1. Let us 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 taking the physical characteristics of the COBOL files into account. For instance, if custCode is defined as a record key in an indexed file, the wrapper should process an indexed access instead of sequential access.
2. Let us assume the following query: Select c.name from customer c, order o where c.ncli=o.ncli; The COBOL DMS does not understand the notion of join. Hence, the wrapper must simulate the query by transforming it into an understandable way for COBOL DMS.
3. Moreover, if the wrapper offers transaction functionalities, it has to ensure the ACID properties since COBOL doesn't support them.

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 are not supported. As a result, 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).

5.3 Wrapper Architecture

In the previous section, we drew up the list of the legacy data system properties and their repercussions on those of wrappers. Three main abstraction levels of wrapper architecture can be inferred from this list:

- *System level.* This level addresses the platform heterogeneity and the autonomy of the legacy database systems.
- *Schema level.* This level focuses on the semantic issues of the legacy databases. It relies on the schema-oriented framework defined in Chapter 4.
- *Query level.* This level considers the query processing of legacy wrappers. Besides the standard language mapping, query processing includes also the schema mapping between two semantically different schemas and semantic integrity control.

5.3.1 System-oriented Architecture

The basic idea is that the legacy databases are not altered; instead they are surrounded by wrappers. Figure 5-7 shows the general framework that overviews the different systems which a wrapper is inserted among. The framework consists of the following components:

- *Legacy applications* that already exist and access the 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 a legacy database through a wrapper.

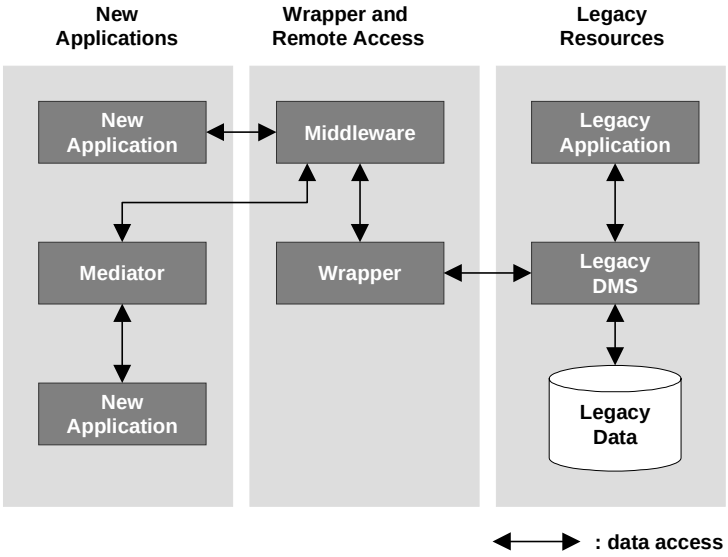


Figure 5-7: System-oriented architecture of a legacy data wrapper.

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.

CORBA [Mowbray, 1995] and RMI [Reese, 1997] are two examples of middleware technologies. CORBA and RMI are two distributed object standards; the first one being 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.

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 and DMS heterogeneity being solved by the wrapper. That is, CORBA allows client applications to communicate with a wrapper without having knowledge of its location, its platform, its language and its network protocol.

New applications

New applications are the software components that access the legacy database through the wrapper interface. A new application can be, among others, a mediator if the system is a federation or an Extract-Transform-Load processor (ETL) for a migration system. The type of the new application strongly influences the wrapper building. For instance, updatable wrappers are often useless in migration processes because the wrapper is only used as a data extractor.

5.3.2 Schema-oriented Architecture

The legacy nature of databases implies that their wrapper must integrate the syntactic homogenization and the semantic layers (Section 5.2.2). Furthermore, if new local applications have to integrate the new functions and requirements of the organization, a wrapper can also include the homogenization layer. As a result, the exported wrapper schema (WES) can correspond to either LH_yS or LH_gS of the schema-oriented framework presented in Chapter 4 (Figure 5-8).

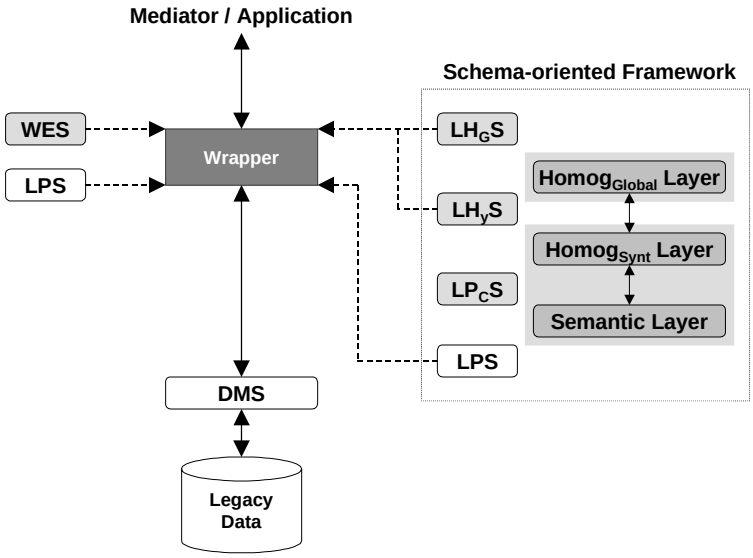


Figure 5-8: Schema-oriented architecture of a wrapper.

For practical reason, we use the logical data model presented in Chapter 2 as the canonical model. We recall that this model has been defined so that it is compliant with standard files and relational model. However, the nature of this model is not relevant to the architecture we are discussing. As shown in Figure 5-9, a model translation (MOD) can be indeed applied in such a way that a schema expressed in the logical data model complies with another canonical data model (Chapter 4, Section 4.2.4).

Wrapper Type	Schema Definition	Schema Semantics
Conceptual wrapper	$WES = MOD_{C \rightarrow W}(LH_yS)$	$\sigma(WES) = \sigma(LH_yS)$
Thick wrapper	$WES = MOD_{C \rightarrow W}(LH_gS)$	$\sigma(WES) = \sigma(LH_gS)$

Figure 5-9: Wrapper types and their schema definition and semantics.

5.3.3 Query-oriented Architecture

One major service of the wrapper is the data and query translation. That is, it translates queries posed on the wrapper schema into commands understandable by the underlying DMS. Moreover, 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 *semantic integrity control* and transaction and failure management if the underlying DMS doesn't support them.

Figure 5-10 shows the successive steps of the wrapper query translation and the result forma-

tion. These steps consist of query analysis, functionality simulation, query translation, integrity control, access plan and processing and finally result formation. We explain and discuss all these steps in the next sections.

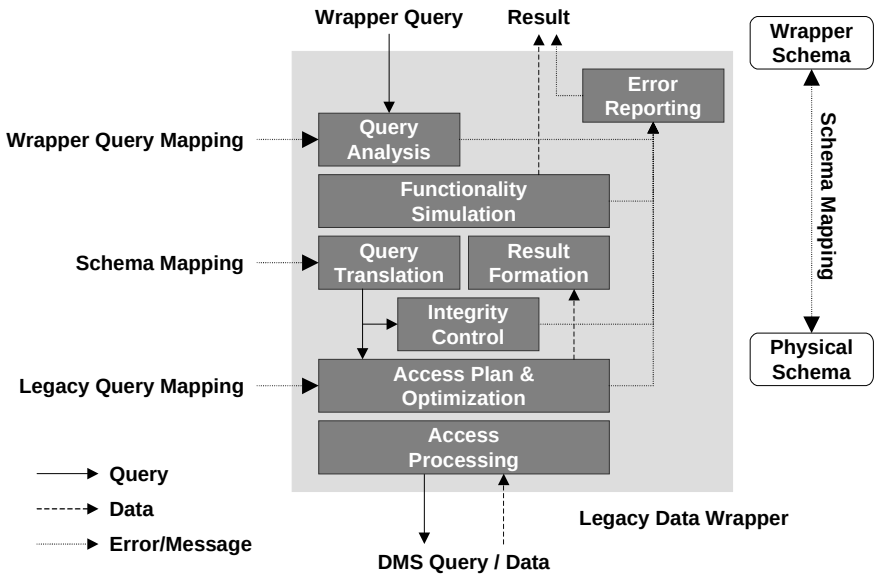


Figure 5-10: Query-oriented wrapper architecture.

5.4 Query Processing

Query processing is the core component of a wrapper. It includes several components of the wrapper architecture, namely, query analysis, query translation, result formation, access plan and optimization and access processing.

5.4.1 Correctness and Efficiency

The main function of the wrapper query processor is to transform wrapper queries into queries understandable by the underlying DMS. Such transformations must be achieved with both *correctness* and *efficiency*:

- *Correctness.* The well-defined mapping from the wrapper schema to the physical schema makes the correctness issue. The mapping formalization by means of schema transformations guarantees the correctness. In Chapter 3, we have described the formal frame-

work of formal transformation which allows the reverse of schema transformation sequence (or history) to be derived automatically. This reversibility allows queries and updates to be automatically translated in either direction between two non-necessarily equivalent schemas.

- *Efficiency.* Producing an efficient execution strategy relies on the legacy query processing capabilities. A same wrapper query can lead to many execution strategies according to the query processing capabilities of the underlying legacy DMS. The access plan, optimization and processing of wrappers use the legacy query processing capability and optimization (Sections 5.4.4 and 5.4.5).

Quality Criteria	Quality Satisfaction
Correctness	Schema transformation formalization of mappings
Efficiency	Query optimization according to the legacy DMS query processing capability

Figure 5-11: Quality criteria and satisfaction of wrapper query processing.

5.4.2 Query Processing Principles

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 using that correspondence to automatically perform the query mappings. Moreover, to make wrappers 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 schema mappings and language mappings.*

Rather than directly translating wrapper queries into legacy queries, we use the query language query defined in Chapter 3 as the bridge for the translation. The idea is to be independent of operational query languages for schema mappings. The schema mappings are therefore applied on a unique generic query language (the query language introduced in Chapter 3).

We can now state the three main successive steps of query translation (Figure 5-10 and Figure 5-12):

- *Wrapper query mappings* ($Q_1 \rightarrow Q_2$): syntactic translation of the query posed on WES into the query language query.
- *Schema mappings* ($Q_2 \rightarrow Q_3$): semantic translation of the query posed on WES to a query posed on LPS using the schema transformation approach presented in Chapter 3, Section 3.5. The schema mappings are defined from the schema transformation sequence WES-to-LPS that defines the mappings between WES and LPS.

- *Legacy query mappings and optimization* ($Q_3 \rightarrow Q_4$): syntactic translation of the query posed on LPS and expressed in query into a query expressed in the DMS query language.

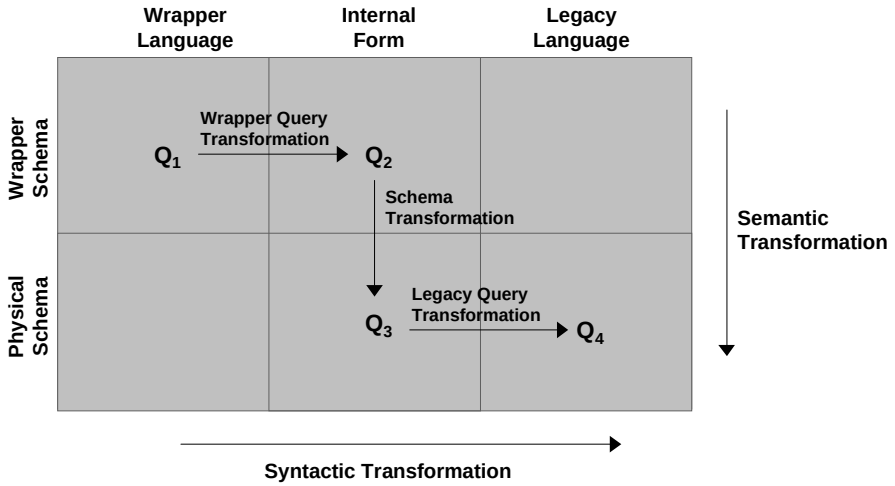


Figure 5-12: Language and schema mappings of a wrapper query Q_1 into a legacy DMS query Q_4.

We first present an informal description of the three steps of query transformation process. We then address the problem of the legacy language mapping and optimization. In this chapter, we do not discuss the other steps of the query transformation process. The schema mapping definition as schema transformations has been already tackled separately in Chapter 3. As for the wrapper language mapping, we make the hypothesis that the wrapper query language and the internal query language (query) are equivalent. The problem of query equivalence is heavily discussed in the database community since the late seventies. We refer to [Graefe, 1993] for a discussion about this problem.

Example

To illustrate the three main steps in query transformation, we use the physical schema (LPS) and the wrapper schema (WES) depicted in Figure 5-13. For simplicity, we use the relational model as the wrapper data model. The differences between LPS and WES can easily be observed in this figure. The sequence WES-to-LPS contains, among others, schema transformation pertinent for the query translation: aggregation of Address.

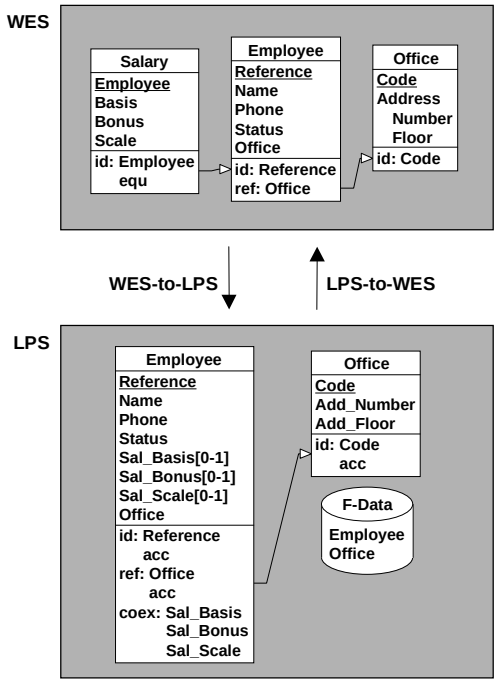


Figure 5-13: Query transformation example based on the schema transformations between the wrapper and physical schemas.

Assume that the query Q_1 is posed on WES, using a SQL-like language as the wrapper query language:

Q_1: select E.Name
from Employee E, Office O
where E.Office = O.Code
and Address.Floor=4;

Step 1. *Wrapper language translation of the query.* The input query is translated into an internal form. In our case, the query Q_1 is translated into an equivalent query using the query formalism. This translation provides a query expressed according to the generic binary data model.

Q_2: [**and**,
[**att**, *Employee, Name,* E, N],
[**att**, *Employee, Office*, E, OO],
[**att**, *Office, Code,* O, OO],

[**att**, *Office, Address,* O, A],
[**att**, *Address, Floor,* A, 4]
]

Step 2. *Schema translation of the query.* This translation is based on the schema transformation used to produce the wrapper schema from the physical schema (e.g., disaggregation of Address). The disaggregation transformation is defined in Figure 5-14 by means of its signature; its structural functions C_ expressed in the schema form and the queries query that state how the extents of each constructs of C_ can be recovered from the extents of the remaining schema constructs C'.

Disagg	(Office, {Add_Number, Add_Floor}) ← Disagg(Office, {Address})
C_(Disagg)	**Query**
[**and**, [**att**, *Office, Address,* OFF, ADD], [**att**, *Address, Number,* ADD, X]]	[**att**, *Office, Add_Number,* OFF, X]
[**and**, [**att**, *Office, Address,* OFF, ADD], [**att**, *Address, Floor,* ADD, X]]	[**att**, *Office, Add_Floor,* OFF, X]

Figure 5-14: Disaggregation transformation definition in the schema form.

The disaggregation is used to automatically translate the queries posed on WES to queries posed on LPS as shown in Chapter 3, Section 3.5.

Q_3: [**and**,
[**att**, *Employee, Name,* E, N],
[**att**, *Employee, Office*, E, OO],
[**att**, *Office, Code,* O, OO],
[**att**, *Office, Add_Floor,* O, 4]
]

Step 3. *Legacy language translation and optimization.* This last step in the translation process translates the internal query into the legacy query language. This query translation is based on the legacy DMS query capability and optimization. Hence, if we consider Oracle as the legacy DMS, the query translation is immediate:

Q_4: select E.Name
from Employee E, Office O
where E.Office = O.Code
and O.Add_Floor = 4;

5.4.3 Wrapper Query Analysis

Before the wrapper query mapping, the first task of a wrapper is the analysis of the wrapper 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 5.4.7). 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 5-13:

```
select Num from Employee 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.

5.4.4 Legacy Query Mapping and Optimization

The process and difficulty of the translation depend on the syntax and expressiveness of both the wrapper (or internal) query language and the legacy query language. If the wrapper query language has a higher expressive power, then either some wrapper queries cannot be translated or they must be translated using both the syntax of the legacy query language and some facilities of the legacy programming language (i.e., the host language of the legacy query language).

Query translation

Depending on whether a source query can be correctly translated into a single target query (or a set of target queries), different techniques can be used to achieve better performance as follows:

- *Case 1. A single legacy query is generated.* If the legacy DMS has a *query optimizer* (e.g., a relational DMS or an object-oriented DMS), then its query optimizer can be used to optimize any correctly translated legacy query. If the legacy system has no query optimizer (e.g., COBOL DMS), then the query translator of the wrapper has to consider the performance of the translated query (see Section 5.4.5).
- *Case 2. A set of legacy queries is needed.* Note that current query optimizers are designed to optimize individual queries. Therefore, they are not capable of globally optimizing a sequence of queries in terms of total processing cost. Two stages of processing multiple queries may be possible to minimize the total cost of during query translation:
    - *Stage 1.* Choose a *good* set of legacy queries or commands that can guarantee correctness. By good we mean that the set of the legacy queries or command should

have certain features that are useful in reducing the total cost. For example, it can pay to access legacy data through index or access keys. This minimizes the number of invocations of the legacy system, and it can also reduce the cost of combining the partial results returned by the queries to form the answer.

- *Stage 2.* Optimize individual queries. This is the same as Case 1.

In Section 5.4.5, we present techniques for query translation by using COBOL as the target system. These techniques comprise wrapper query decomposition and optimization of the individual queries.

Query optimization

An important aspect of legacy wrapper mapping is query optimization. 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.

The main factor affecting the performance of an execution strategy is the size of the intermediate result sets that are produced during the execution. This estimation is based on *statistical information* about the base constructs and formulas to predict the cardinalities of the results of the operations. There is a trade-off between the precision of the statistics and the cost of managing them, the more precise statistics being the more costly to collect. For a discussion on the statistics acquisition and use, we refer to [Elmasri, 1994].

We can also use heuristic rules based on the physical data access. *Heuristic rules* are used for ordering the operation in 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, in the relational algebra, a main heuristic rule states that Select and Projection operations should be applied before the join and other binary operations. The reason is that Select and Projection operations usually reduce the size of intermediate files.

Example

Let us consider an application that issues a selection query to a wrapper. The underlying legacy information data it accesses are recorded in COBOL files. Let us assume that the three queries of Figure 5-15 are processed by the wrapper. Each query includes a different selection statement: the first one is made up of an attribute that is a COBOL record key; the second one, an attribute that is a COBOL alternate record key; and the last one, an attribute on which no key has been implemented.

The wrapper processes the queries of Figure 5-15 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 5-15, because A1 is defined as a

record 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.

Query Selection	Legacy Physical Access	Access number
[**att**, *A, A1,* a, "A"]	A1 is a record key (and an access key)	1
[**att**, *A, A2,* a, "AB"]	A2 is an alternate record key with duplicates (an access key)	>=1
[**att**, *A, A4,* a, "ABC"]	sequential	>=1

Figure 5-15: Wrapper query and legacy physical access.

5.4.5 COBOL Query Mapping

As noted previously, the legacy DMS vary greatly in their query processing capabilities. As a result, there are as many legacy query processors and optimizers as legacy DMS families and even as versions of the same DMS. We can't therefore be exhaustive and present a solution for each existing DMS family. We rather propose to address some aspects of the legacy query mapping by concentrating our attention on the COBOL DMS.

COBOL DMS does only handle mono-entity primitives. These primitives can be categorized as navigation commands and update commands. Update commands are used to insert or delete instances of a record type. Navigation commands retrieve one instance of the record type. COBOL allows the programmer to start a sequence based on an indexed key (start-key and read-key), then to go on this sequence through read-next primitives. In Figure 5-16, we classify the COBOL commands according to their access type: indexed unique access; indexed non unique access and sequential access. For simplicity and genericity, these commands are written in a pseudo-language.

Access Pattern	Procedural Pattern (pseudo code)	Comments
Indexed unique access (A1 is a record key)	read A(A1=*a1*);	read represents the reading of the record of type A identified by a field value (*a1*)
Indexed non-unique access (A2 is an alternate record key with duplicates)	read-first A(A2=*a2*); read-next A(*AA*, A2=*a2*);	read-first represents the reading of the first record of type A identified by a field value (*a2*); read-next represents the reading of the record next to the current record *AA* of type A identified by a field value (*a2*)
Sequential access	read-first A(); read-next A(*AA*);	read-first represents the reading of the first record of type A; read-next represents the reading of the record next to the current record *AA* of type A
Delete	delete A(*AA*);	delete represents the deleting of the record *AA* of type A
Insert	insert A(*AA*);	delete represents the insertion of the record *AA* of type A

Figure 5-16: Access patterns and procedural patterns close to the COBOL primitives.

We hereafter address the problem of the query mapping and optimization of an internal query expressed in query into a set of legacy access primitives.

Query decomposition

The internal query (query) is decomposed into a sequence of legacy access primitives that can be performed by the COBOL DMS. The query decomposition is based on the access primitives to a single entity type (or entity record). The algorithm produces a legacy access query plan that only executes mono-entity access operations and tries to minimize the sizes of intermediate results in ordering binary (multi-entities) operations.

We hereafter illustrate the algorithm principles by examining queries posed on the COBOL physical schema depicted in Figure 5-17.

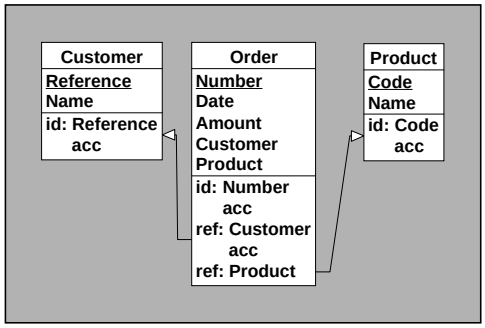


Figure 5-17: Cobol physical example of schema. For readability, we have specified the reference groups to the schema.

Let us denote by $q_{i-1} \Rightarrow q_i$ a query q decomposed into two subqueries, q_{i-1} and q_i, where q_{i-1}, is executed first and its result is consumed by q_i. Given an n-entity query q, the query translator decomposes q into n mono-entity subqueries $q_1 \Rightarrow q_2 \Rightarrow ... \Rightarrow q_n$. To represent this query plan in terms of query, we introduce two constructs:

- [**monoentity**, $ETname_{input}$, $ETname_{output}$, eq, query] identifies a mono-entity query query that consumes the instances of $ETname_{input}$ and returns the instances for $ETname_{output}$. The instances of $ETname_{input}$ are linked to those of $ETname_{output}$ by means of the construct eq.
- [**nentity**, query] identifies a n-entity query.

The query decomposition uses two basic techniques: *decomposition* and *reduction*. These techniques are presented and illustrated in the rest of this section.

Decomposition. Decomposition is the first technique used by the legacy query translator. It breaks a query q into $q' \Rightarrow q''$, based on a common entity type variable that is the result of q'. If the query q expressed in query includes at least one selection on attributes:

[**att**, *ETname, Att1Name*, ET, value1]

where value1 is a value of the attribute Att1Name and ET is a variable of the entity type ETname.

then, the query can be decomposed into two subqueries, q' followed by q'', by grouping in q' the schema constructs that has the same ET as variable.

Let us assume q expressed in the language Query is of the form:

q: [**and**,
[**att**, *ETname, Att2Name,* ET, A2],
[**att**, *ETname, Att1Name,* ET, value1],
[q-end]]

where A2 is a variable, value1 is a value and q-end is a query involving entity type variable other than ET.

q': [**and**,
[**att**, *ETname, Att2Name,* ET, A2],
[**att**, *ETname, Att1Name*, ET, value1]]

q'': [**and**,
[**att**, *ETname, Att2Name,* ET, A2],
[q-end]]

The query plan indicates that q' is first executed and returns instances of ET:

[**plan**,
[**monoentity**, _, *ETname*, _, q'],
[**nentity**, q'']
]

The construct monoentity identifies a mono-entity query and can therefore be processed by the legacy DMS. monoentity is translated into the legacy procedural patterns of Figure 5-16. A simple heuristic rule is used for ordering the priority physical access:

record key (unique index) 〉 alternate record key (non unique index) 〉 other

This order priority defines the primary access of the sequence. If more than one selection attribute is defined in monoentity, then we must verify if the current instance satisfies the other selection attribute condition. Figure 5-18 presents some typical access patterns.

Mono-Entity query	Access Pattern	Procedural Pattern
[**att**, *Order, Number*, OO, R]	unique key access	return **read** Order(Number= *R*));

[**and**,
[**att**, *Order, Customer,* OO, C],
[**att**, *Order*, *Date,* OO, D]]
	non unique access key and filter	output ← ∅;
		OO ← **read-first** Order(Customer=C)
		while found then
		if (OO.Date=D) then
		output ← output ++ OO;
		OO ← **read-next** Order(OO, Customer=C);
		end-if;
		end-while;
		return output;

[**att**, *Order, Date,* OO, D]	sequential access	output ← ∅;
		OO ← **read-first** Order();
		while found then
		if (OO.Date=D) then
		output ← output ++ OO;
		OO ← **read-next** Order(OO);
		end-if;
		end-while;
		return output;

Figure 5-18: Mono entity query and their procedural patterns (++ means an append).

The query plan indicates that q' is first executed and returns instances of ET:

[**plan**,
[**monoentity**, _, *ETname*, _, q'],
[**nentity**, q'']
]

Example

To illustrate the decomposition technique, we apply it to the following query.

q_1: [**and**,
[**att**, *Customer, Name,* CC, X],
[**att**, *Customer, Reference,* CC, RE],
[**att**, *Order, Customer,* OO, CU],
[**eq**, CU, RE],
[**att**, *Order, Product,* OO, PR],
[**att**, *Product, Code,* PP, CO],
[**eq**, CO, PR],
[**att**, *Product, Name,* PP, "Book"]]

After decomposition of the selection query, query q_1 is replaced by q_{11} followed by q_{12}:

q_{11}: [**and,**
[**att**, *Product, Code,* PP, CO],
[**att**, *Product, Name,* PP, "Book"]]

q_{12}:[**and**,
[**att**, *Customer, Name,* CC, X],
[**att**, *Customer, Reference,* CC, RE],
[**att**, *Order, Customer,* OO, CU],
[**eq**, CU, RE],
[**att**, *Order, Product,* OO, PR],
[**att**, *Product, Code,* PP, CO],
[**eq**, CO, PR]]

These queries are inserted into the query plan as follow:

[**plan**,
[**monoentity**, _, *Product*, _, q_{11}],
[**nentity**, q_{12}]
]

The succesive decomposition of q_{12} generates:

q_{121}: [**and**,
[**att**, *Order, Customer,* OO, CU],
[**att**, *Order, Product,* OO, PR],
[**att**, *Product, Code,* PP, CO],
[**eq**, CO, PR]]
q_{122}:[**and**,
[**att**, *Customer, Name,* CC, NA],
[**att**, *Customer, Reference,* CC, RE],
[**att**, *Order, Customer,* OO, CU],
[**eq**, CU, RE]]

Thus query q_1 has been reduced to the subsequent queries $q_{11} \Rightarrow q_{121} \Rightarrow q_{122}$. Query q_{11} is mono-entity and can be performed by the COBOL processor. However, q_{121} and q_{122} are not mono-entity queries and cannot be reduced by the mechanism described above.

Using the rules developed so far, we are able to translate an internal query involving a single entity type into a legacy procedural pattern. These rules alone have a limited use since a query can involve more than one entity type. Let us discuss now this kind of legacy query mappings.

Reduction. Multi-entity queries cannot be detached by means of the above rules. Multi-entity queries are converted into mono-entity queries by reduction. Given an n-entity type query q,

query reduction proceeds as follows. First, one entity type in q is chosen for query reduction. Let ET_1 be this entity type. Then, the constructs in q that have ET_1 as variable are replaced by their actual values of ET_1, thereby obtaining a (n-1)-entity query. Query reduction can be summarized as follows:

query(ET_1, ET_2, ..., ET_n) *is reduced into* query'(ET_2, ..., ET_n)

Example

To illustrate the reduction technique, we apply it to the query q_{121}:

q_{121}: [**and**,
[**att**, *Order, Customer,* OO, CU],
[**att**, *Order, Product,* OO, PR],
[**att**, *Product, Code,* PP, CO],
[**eq**, CO, PR]]

The entity type Product is common to q_{11} and q_{121}. We therefore use it as the reduction basis. The attribute Product of Order referred to by in Product is replaced by its actual values using the eq construct:

$q_{121'}$: [**and**,
[**att**, *Order, Customer,* OO, CU],
[**att**, *Order, Product,* OO, PR]]
where PR is instantiated with values of CO defined in q_{11}([**eq**, CO, PR]).

This query is then inserted in the query plan:

[**plan**,
[**monoentity**, _, *Product*, _, q_{11}]
[**monoentity**, *Product, Customer*, [**eq**, CO, PR], $q_{121'}$]
[**nentity**, q_{122}]
]

Algorithm. The legacy query translation algorithm is depicted in Figure 5-19. The algorithm works recursively until no multi-entity queries remain to be analyzed. The multi-entity queries are processed by reduction. It consists of decomposing the input query into mono-entity queries that can be processed by the legacy DMS. Each mono-entity query is inserted into the query plan. For a multi-entity query, the smallest mono-entity query result whose cardinality is known is chosen for substitution. This simple method enables one to generate an efficient query access plan. The result of the algorithm is a query plan of mono-entity queries that can be processed by the underlying DMS.

```
LegacyAcccesPlan(query q)

    MonoEntityQuery ⟵  IsMonoEntityQuery(q);

    if MonoEntityQuery then
        // define the mono-entity construct:
        monoentity ⟵  define monoentity construct (q);
        insert monoentity into the query plan;
    else begin
        // query decomposition:
        // separate query into one mono-entity defined on the same entity type
        // and one n-entity query defined on the other entity type(s):
        qMono, qN ⟸  q;

        // define the mono-entity construct
        monoentity ⟵  define monoentity construct (qMono);
        insert monoentity into the query plan;

        ET ⟵  Choose_Variable_ET(qN);
        qN' ⟵  reduce qN by taking the constructs that have ET as variable;
        LegacyAccessPlan (qN'); // Recursive call
    end-if;
end.
```

Figure 5-19: Algorithm of the query decomposition

Algorithm issues

So far, we have presented the principles of the multi-entity query decomposition. We now discuss some basic algorithm issues related to how an application program effectively accesses the legacy data through a wrapper. An application program typically goes through the result instances and processes them one at a time. The concept of *cursor* or *current* is then used to allow instance-at-a-time processing.

Current management. The current indicator plays the role of position holders so that we can keep track of the instances most recently accessed. Each current indicator can be thought of as a record pointer that points to an instance. As far as wrapping is concerned, there are three kinds of currents: the current of the underlying DMS (here COBOL), the current of the wrapper and the current that the program application uses to communicate with the wrapper. These current indicators can be identical but are generally different:

- *The current of the application program*: The synchronization of the currents of a program application and those of the DMS are managed by the wrapper through its current.
- *The current managed by COBOL:* In COBOL, retrievals and updates are handled by moving or navigating through the record instances. For each record type, COBOL automatically keeps track of the most recently accessed record of that record type.
- *The current managed by the wrapper:* For each wrapper query, the wrapper keeps track of the most recently accessed occurrence from the wrapper query. The representation of the current indicators depends on the wrapper query posed on WES and its correspond-

ing legacy query (posed on LPS). The current indicator is therefore represented by the composition of the explicit identifiers (RECORD KEY) of all the record types[2] (RC) involved in the legacy query. Moreover, if the entity type used as variable in the wrapper query is represented by a multivalued attribute (array in COBOL) in LPS, then the current indicator is not only represented by the identifier of the record type that comprises the multivalued attribute but also by the position of the most recently accessed instance in the file.

Figure 5-20 summarizes the three main different representations of the wrapper current indicators for a wrapper query that has only one entity type ET as variable:

- *Case 1.* ET corresponds to only one RC: we use the explicit identifier of RC.
- *Case 2.* ET corresponds to more than one RC: we use the composition of the explicit identifiers of all the RC involved.
- *Case 3.* ET corresponds to an array within a RC: the current indicator is represented by the explicit identifier of RC and the position of the most recently accessed position of the array[3].

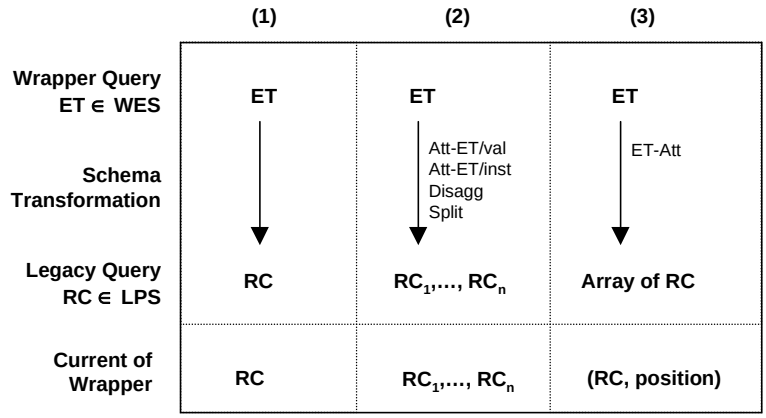


Figure 5-20: Representations of the current indicator of a wrapper for a wrapper query with one entity type ET as variable. The correspondences between WES and LPS are illustrated by means of schema transformation types.

Each time a program executes a first or next statement, the current indicators affected by the query is updated by the wrapper. A clear understanding of how wrapper statements both ef-

2. Recall that an entity type in the generic data model is called a record type in COBOL (Chapter 2, Section 2.3).
3. For simplicity, we only consider one level of arrays.

fect and depend on the current indicators managed by COBOL and the wrapper is necessary. In the next subsections, we discuss and illustrate how the different wrapper statements affect the currents managed by COBOL.

Instance-by-instance access of a mono-entity legacy query. In Figure 5-16, we presented the legacy procedural patterns that process the optimized physical accesses of mono-entity legacy queries. As already stated, a wrapper provides an instance-by-instance access. To provide such an access, the legacy procedural patterns are revised and combined to form two access types that return an instance of a record type at a time: Read-First-*RecordName* and Read-Next-*RecordName*. The forms of these instance-by-instance access patterns are given in Figure 5-21.

Record Access Interface	Comments
Read-first-*RecordName* Input: selection Output: record	Retrieves the first record of the specified type (RecordName) that satisfies the condition expressed by the selection[a]. If a record is found, record is returned; null otherwise.
Read-Next-*RecordName* Input: selection, current Output: record	Searches the current record (current) by using the access key of the record type. Searches forward in the record type to retrieve a record that satisfies the condition expressed in the selection. If a record is found, record is returned; null otherwise.

a. selection ::= **(***AttributeName* op *value***)** {, **(***AttributeName* op *value***)**} *where AttributeName* is the name of an attribute of the specified record type and op is an operator.

Figure 5-21: Record access interface: read-next and read-first.

From these access patterns, we can state the flows of the current indicator between the wrapper and COBOL (Figure 5-22):

- For a *read first* statement, the current of a record type retrieved by COBOL becomes the wrapper current for this record.
- For a *read next* statement, the wrapper current is first used to set the current indicator of COBOL. This allows to locate the record that corresponds to the wrapper current and to begin the retrieve from this record. Once the next record or the next position of the array is retrieved by COBOL, the current indicator of the wrapper is updated.

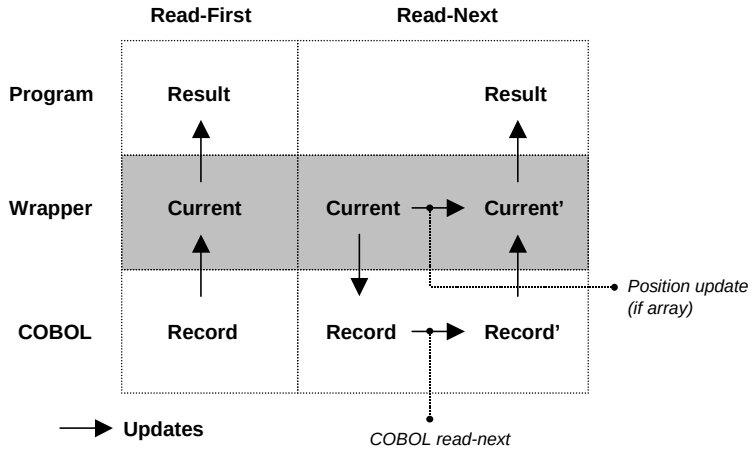


Figure 5-22: Mono-entity current management between COBOL and the topped wrapper. In a first statement, the record instance retrieved by COBOL is copied in the current managed by the wrapper. In a next statement, the current managed by the wrapper is used to retrieve its next current: either for retrieving the next record in the file; or for accessing the next position in the array.

Instance-by-instance access of N mono-entity legacy queries. Mono-entity queries form a query plan that results from the schema mapping (Section 5.4.2) and the decomposition algorithm (Section 5.4.5). Since the query planning does not provide an instance-by-instance access, it must be revised. To illustrate our discussion, we take up the previous example of query planning resulting from the query decomposition (Figure 5-24).

```
[plan,
    [monoentity, _, Product, _ [and,
                        [att, Product, Code, PP, CO],
                        [att, Product, Name, PP, "Book"]]],
    [monoentity, Product, Order, [eq, CO, PR],
                        [and,
                            [att, Order, Product, OO, PR],
                            [att, Order, Customer, OO, CU]]]
    [monoentity, Order, Customer, [eq, CU, RE],
                        [and,
                            [att, Customer, Reference, CC, RE],
                            [att, Customer, Name, CC, X]]],
]
```

Figure 5-23: Query planning.

The query planning is written in a procedural form in Figure 5-24.

```
Output ← ∅;
PP: ← Read-first-Product(Name="Book");
While PP ≠ null do
    OO ← Read-first-Order(Product=PP.Code);
    while OO ≠ null do
        CC ← Read-first-Customer(Reference=OO.Customer);
        If OO ≠ null then
            Output ← Output ++ CC.Name;
            end-if;
        OO ← Read-next-Order(Product=PP.Code, OO);
    end-while;
    PP ← Read-next-Product(Name="Book", PP);
end-while;
return Output;,
```

Figure 5-24: Access plan resulting of the query decomposition

However, the algorithm of Figure 5-24 does not offer an instance-by-instance access. The next statements are concealed inside the loops. We therefore transform this algorithm in order to provide the first and next statements (Figure 5-25):

- The *first statement* starts searching sequentially through the record types involved in the search until it finds the first record occurrence of Customer. Note that the first statement section of the algorithm only uses the current records kept track by the DMS. These currents set those management by the wrapper (see below).
- The *next statement* starts assigning the currents of the involved record types to those managed by the wrapper. So that, the current indicators of the DMS correspond to the latest records accessed by the algorithm. The search of the next statement can then begin.

```
FETCH
input: current, type
output: result, current

result ← null;
if type = FIRST then goto FIRST;
if type = NEXT then goto NEXT;

FIRST:    PP ← Read-first-Product(Name="Book");
TEST-O:   if PP = null then goto END-P; end-if;
          OO ← Read-first-Order(Product=PP.Code);
TEST-C:   if OO = null then goto END-O; end-if;
          CC ← Read-first-Customer(Reference=OO.Customer);
          if CC = null then goto END-C; end-if;
          result ← CC.Name;
          current.Product ← PP.Code;
          current.Order ← OO.Number;
          current.Customer ← CC.Reference;
          goto EXIT;
```

```
NEXT:
        PP ← Read-first-Product(Code=current.Product);
        OO ← Read-first-Order(Number=current.Order);

END-C:  OO ← Read-next-Order(Product=PP.Code, OO);
        goto TEST-C;
END-O:  PP ← Read-next-Product(Name="Book", PP);
        goto TEST-O;
END-P:
EXIT:  return result;
```

Figure 5-25: Operational algorithm.

By transforming the access structure of the algorithm, we obtain an algorithm that has the two following properties:

- The algorithm does not require a state memorization of the control structures;
- The algorithm can be executed in a procedural environment.

5.4.6 Semantic Integrity Control

Another important and difficult problem for a wrapper is how to guarantee the *data consistency*. A legacy database is said to be consistent if its contents satisfy a set of constraints that can be either explicit or implicit (see Chapter 4, Section 4.2.3). Maintaining a consistent database therefore requires mechanisms that ensure that wrapper updates satisfy the set of explicit and implicit integrity constraints.

Explicit and implicit constraint

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 (Figure 5-26). A major difficulty in designing an integrity subsystem of a wrapper is finding efficient enforcement algorithms.

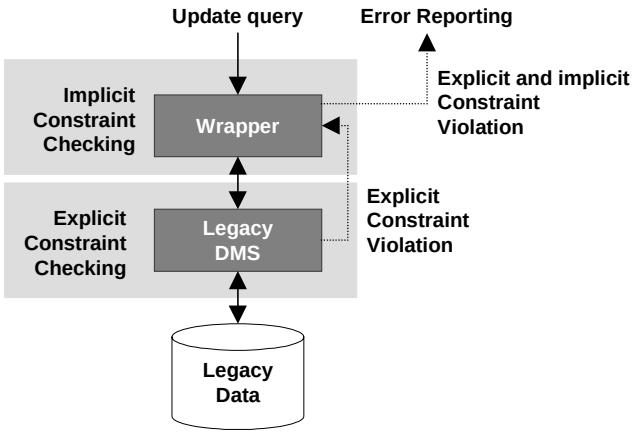


Figure 5-26: DMS and Wrapper management of explicit and implicit constraints.

Methods of inconsistent update rejections

Two basic methods permit the rejection of inconsistent updates [Özsu, 1991]. The first one is based on the *detection of inconsistencies*. The update u is executed, consisting of a change of the database state D to D_u. The wrapper enforcement algorithm verifies, by applying tests derived from these implicit constraints, that all relevant implicit 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 updating the database state, they are generally called *post-tests*. This approach can be inefficient if 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 wrapper enforcement algorithm verifies that all relevant implicit constraints will hold after updating those instances. This is generally done by applying to those instances test that are derived from the implicit 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 wrapper development 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 the wrapper development 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 <ET, T, C> 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 checkings can 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

Let us consider the wrapper schema of the Figure 5-27. This schema is made up of two entity types A and B and contains three constraints. Two of them are implicit: the reference constraint and the identifier of the entity type A. These are the results of two semantic-augmenting transformations: namely, create-Identifier and create-Reference.

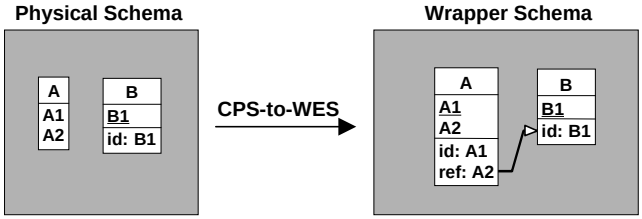


Figure 5-27: Wrapper schema example that illustrates an implicit reference constraint and an implicit identifier.

Let us consider the implicit constraint checking assertions associated with the reference constraint:

<A, INSERT, C1> and <B, DELETE, C2>

where C1 is ∀ NEW ∈ (new instances of A), ∃ b ∈ (instances of B): NEW.A2 = b.B1;

and C2 is ∀ a ∈ (instances of A), ∀ OLD ∈ (deleted instances of B): a.A2 ≠ OLD.B1.

Let us now illustrate the constraint checking algorithm (Figure 5-28). 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 with each ET by implementing its constraint checking assertions. The second step consists in performing the update itself if no implicit constraint is violated.

```
Entity type and update type: <ET, T>
Checking Assertions:         {<Ci>}
Instance:                    et

// implicit constraint checking:
    inconsistency ←  false;
    for each compiled assertion Ci do
        result ←  retrieve all new (respectively old) instances of ET where ¬ (Ci)
        if card(result) ≠ 0 then
            inconsistency ←  true;
            exit;
        end-if;
    end-for;

// explicit constraint checking:
    if not (inconsistency) then
        submit to the DMS the update <ET, T> with respect to et;
        result ←  receive the update result (0 is a success);
        if (result ≠ 0) then
            inconsistency ←  true;
        end-if;
    end-if;

    if inconsistency then
        reject the update;
    end-if;
```

Figure 5-28: Algorithm of the implicit and explicit constraint checking.

Example

Let us consider the example of the implicit reference constraint of Figure 5-27. The main constraint checking assertions associated with this constraint are summarized in Figure 5-29. 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 not found then *inconsistency* ← true; end-if;
B	DELETE	read-first A(A2=B.B1) if found then *inconsistency* ← true; end-if;
Entity Type	**Update Type**	**SQL-like Expressions**
A	INSERT	if not exists (select * from B where B.B1= A.A2) then *inconsistency* ← true; end-if;
B	DELETE	if exists (select * from A where A.A2=B.B1) then *inconsistency* ← true; end-if;

Figure 5-29: The constraint checking assertions associated with a reference constraint.

Checking assertion enforcement order

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.

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. If no statistics information is available, we can only use heuristic rules for ordering the checking assertion enforcements using the physical access plan. We push the checking assertions that only access data by means of index (or access key) at the top of the list so that they are evaluated as early as possible.

5.4.7 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 return 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 error can result from any 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 5.4.3) or when it performs the semantic control of an implicit constraint (see Section 5.4.6).

Besides the error codes it receives or detects, a wrapper 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 it in a different manner, using different error numbers, message types, programming styles. A wrapper must therefore simplify error information processing 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.

5.4.8 Additional Functionality

Up to this point, the basic feature of a wrapper we have considered is the retrieval and update query processing. In Sections from 5.4.1 to 5.4.5, we discussed how wrapper queries can be processed and optimized and, in Section 5.4.6, we explained how a wrapper can guarantee the legacy data semantic integrity in presence of implicit constraints.

However, we never considered what happens if, for example, two wrapper queries attempt to update the same legacy data; or if a failure of the legacy system or of the wrapper occurs during execution of a query; or if an unauthorized action is performed. Maintaining a consistent legacy database requires that the wrapper controls not only the semantic integrity but also the concurrence, the reliability and the security. Although these kinds of functionality are somewhat beyond the scope of this thesis, we discuss them briefly for the sake of completeness.

Security

Definition and concept evolution. Security is a method to maintain accountability and control of the 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.

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 Internet) 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 aspects: 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 exchange 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

In our schema-oriented approach, the wrapper mappings are defined from schema transformations between two non-necessary equivalent schemas (Chapter 3, Section 3.5.3). The wrapper schema can be therefore defined as a subset of the physical schema and be used to hide sensitive data from unauthorized users.

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. Transaction management can be viewed in two dimensions: autonomy and heterogeneity.

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

Heterogeneity. Each legacy DMS family can 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 minimal effort. Some older DMS do not 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 sophisticated 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 transaction 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 legacy DMS 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 [Özsu, 1991].

Example

A *memoryless wrapper* is a wrapper that keeps no context handle of requests that it performs. A context handle holds all the information required by the wrapper for performing a query (for instance, the current indicators). 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 track of what has happened between this client and the wrapper.

A memoryless wrapper is not required to maintain contextual information about the client applications. Each request from a client application contains the context handle needed to satisfy the request. To submit a query, a client application sends its request and its context handle to the wrapper. The wrapper uses the context handle to identify the client and to prepare the request processing. It updates the context handle according to the

request results and then returns it to the client application.

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

5.5 Wrapper Development

This section deals with developing wrappers for legacy data systems. Developing wrappers, as we have seen through the previous sections, is an extensive and complex engineering activity[4] that involves experiment and method, thus making necessary the facilitation of the wrapper development/construction.

Our approach of wrapper development addresses the challenge of DMS legacy and heterogeneity by proposing a generic approach for the wrapper development while taking a schema transformation approach to mapping definition. As a result, the wrapper architecture of Figure 5-10 is refined such that it can be instantiated for any legacy data models and systems. 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 section attempts to answer the wrapper development systematically. Our objective is to provide a generic framework of the wrapper development which can be modified, extended, and customized depending on the specific needs. The CASE environment for the wrapper generation will be presented separately in Chapter 7.

The section is organized as follows. Based on our experience in data wrapping, Section 5.5.1 presents the main baselines of the wrapper development. In Section 5.5.2, we design the wrapper architecture such that it can be instantiated for any legacy data models and systems. Section 5.5.3 presents some metrics of the wrapper development cost. They illustrate the necessity of the facilitation of the wrapper development/construction.

4. [Elmagarmid, 1986] points out that typically the construction of a wrapper requires "6 month-work".

5.5.1 Main Baselines

Observations

In the InterDB project, we have built hard-coded wrappers for several legacy data systems, including relational databases and COBOL files (Section 5.6). 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. In Chapter 3, we demonstrated that the query 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 our experiences in application project areas of city administration systems [Thiran, 2002d], 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.

Wrapper dimensions

Based on these observations, we define three dimensions of a wrapper dedicated to legacy data sources (Figure 5-30): (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.

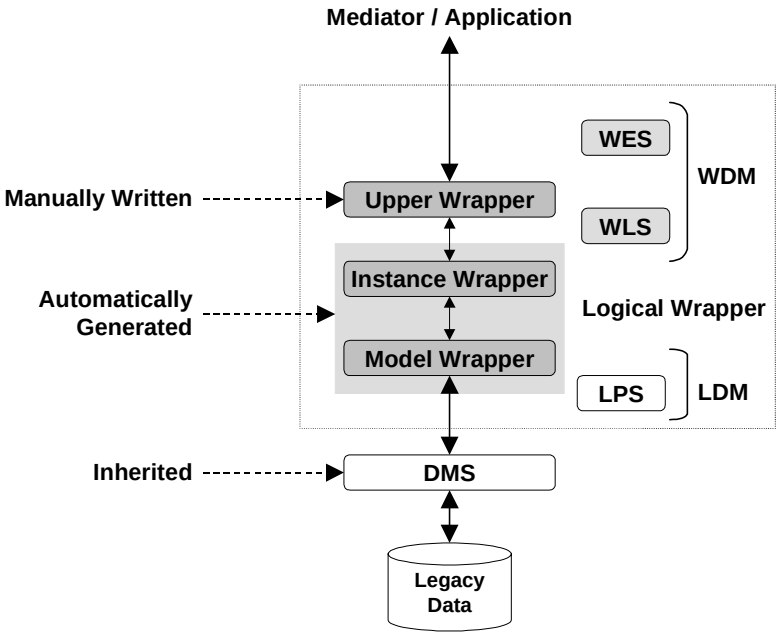


Figure 5-30: The three dimensions and schemas of a wrapper.

In the next sections, we will describe the three wrapper dimensions distinctly though they can be implemented in a unique software component.

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 schema 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 2, and a realistic set of mapping rules, such as those presented in Section 3.4 of Chapter 3.

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 mapping rules. The logical wrapper can be instantiated in two levels (Figure 5-31):

- *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.

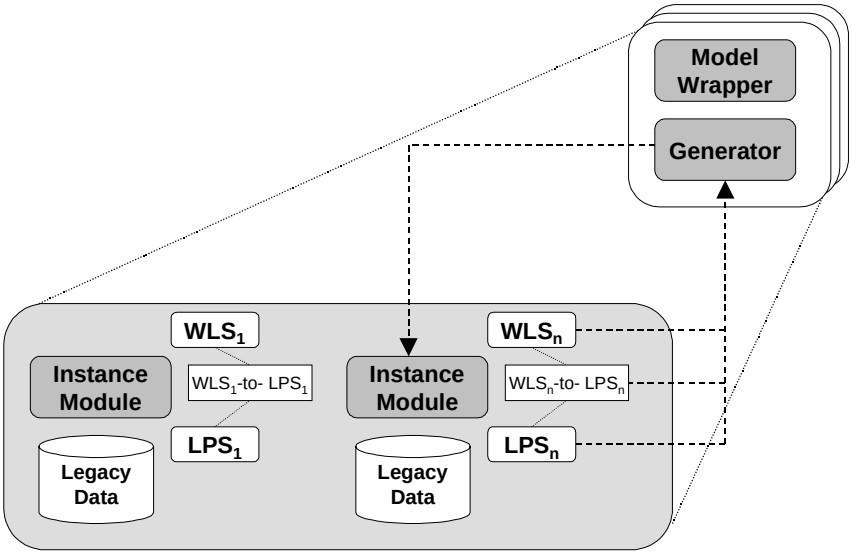


Figure 5-31: Logical wrappers: model wrappers and instance wrapper generators dedicated to a DMS family.

Upper wrapper. The upper wrapper offers the export schema (WES) of the wrapper schema. This view is based on the same logical data model. It manages the complex mapping rules that cannot be taken into account by the logical wrapper. It is therefore programmed manually. An example of such complex mapping rules are the user-defined functions whose semantics are known by the user only (Chapter 3, Section 3.4 and Chapter 6, Section 6.3.3). They have no predefined t mapping part, so that they cannot be formally processed.

5.5.2 Wrapper Component Generation

Schema and mapping definition

The wrapper models include the legacy data models supported by the legacy databases and the wrapper logical model. All these models have been defined in Chapter 2. We recall that they are built as specializations of the generic model. The wrapper logical model is intended to describe all the structures and constraints that exist explicitly and implicitly in all the legacy data models.

Based on the schema hierarchy, we can state the relationships at the schema level:

$\sigma(WLS) = \sigma(LPS) \cup \nu$ (instance wrapper)

$\sigma(WES) = \sigma(WLS) \cup \nu'$ (upper wrapper)

σ (WES) = σ (LPS) $\cup$ ν $\cup$ ν' (whole wrapper)

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

Referring to the schema-oriented framework defined in Chapter 4, the extra semantics can also be expressed as follows:

$\nu = \nu' = \varnothing$	if the wrapper includes the syntactic homogenization layer only;
σ (LPS) $\cup$ ν $\cup$ ν' = σ (LH_yS)	if the wrapper includes the semantic and syntactic homogenization layers;
σ (LPS) $\cup$ ν $\cup$ ν' = σ (LH_gS)	if the wrapper includes the semantic, syntactic and global homogeinzation layer.

ν and ν' are the result of schema transformation sequences between the schemas of the hierarchy of the wrapper modules:

ν : WLS=LPS-to-WLS(LPS)

ν': WES=WLS-to-WES(WLS)

The instance wrapper emulates the query mappings defined from LPS-to-WLS whereas the upper wrapper takes in charge the other query mappings. Any schema transformation of <LPS-to-WLS WLS-to-WES> is defined as an instantiated form of a generic transformation defined at the generic data model.

Let us consider Γ, the set of generic transformations defined for the generic data model. The *instance wrapper generator* is defined for a predefined set of generic schema transformations Γ_{LW} which a predefined mapping pattern is attached to:

$$\Gamma_{LW} \subseteq \Gamma$$

Hence, the instance wrapper can emulate the instances of any generic schema transformation of Γ_{LW}:

$\forall$ T $\in$ LPS-to-WLS, T is defined in Γ_{LW}

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

Γ_{UW} : $\Gamma_{UW} \cup \Gamma_{LW} = \Gamma$ and $\forall$ T $\in$ WLS-to-WES, T is defined in Γ_{UW}

Finally, we can refine the schema transformation sequence LPS-to-WES that is made up of schema transformations of LPS-to-WLS and WLS-to-WES:

$$\forall\, T_i \in \text{LPS-to-WES},$$
$$(\,(T_i \in \text{LPS-to-WLS} \Rightarrow T_i \in \Gamma_{LW}\,)$$
$$\wedge\,(T_i \in \text{WLS-to-WES} \Rightarrow T_i \notin \Gamma_{LW}\,)\,)$$

Wrapper components and their definition

The wrapper architecture shown in Figure 5-32 provides a generic wrapper framework independent of a particular legacy database and of a DMS family. Some parts represent the model wrapper that is built once for a DMS family. A generator computes the other parts for a particular legacy data system. They are built from the results of the methodology processes that will be discussed in Chapter 6.

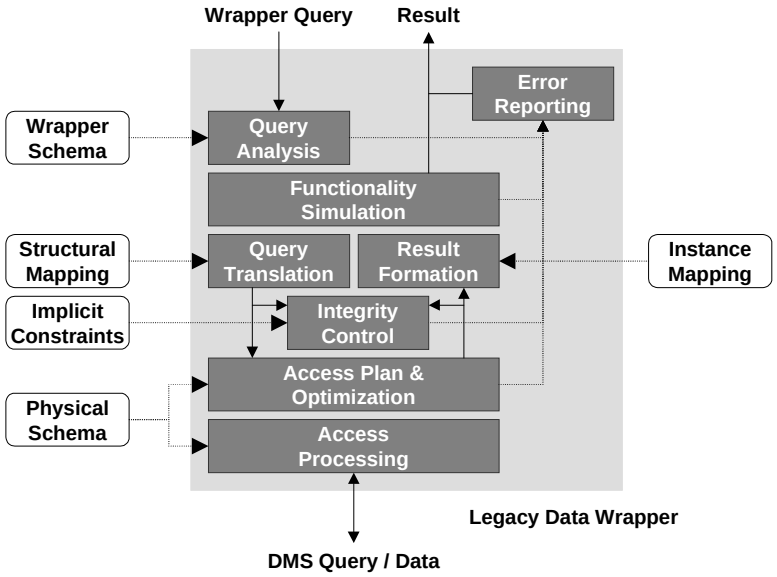


Figure 5-32: The components of the basic and upper wrappers.

Model wrapper

The model wrapper is made up of components specific to a DMS family, and is unique for all the wrappers related to a DMS family. As already stated, 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 Oracle V5. We assume that the wrapper offers a SQL-like interface to the new applications. We also assume that

the wrapper allows update queries and the use of the two phase commit protocol. Figure 5-33 compares the complexity of the model wrappers according to the services they emulate and the underlying DMS.

Wrapper Services	Cobol	Oracle V5
Wrapper query analysis	yes	yes
Wrapper query mappings	yes	yes
Schema mappings	yes	yes
Legacy query translation	yes	yes
Query optimization and processing	yes	no (delegated to Oracle V5)
Cursor concept	yes	no (delegated to Oracle V5)
Error reporting	yes	yes

Figure 5-33: Model wrappers and the services they emulate according to the underlying DMS.

Instance and upper wrappers

Both the instance and upper wrappers are modeled through schema transformations LPS-to-WES. We recall that LPS-to-WES has been defined as a sequence of two minimal sequences: LPS-to-WLS and WLS-to-WES such that:

$$\forall T_i \in \text{LPS-to-WES},$$
$$((T_i \in \Gamma_{LW} \Rightarrow T_i \in \text{LPS-to-WLS})$$
$$\wedge (T_i \notin \Gamma_{LW} \Rightarrow T_i \in \text{WLS-to-WES}))$$

On the one hand, LPS-to-WLS is the schema transformation sequence emulated by the instance wrapper and that only includes the transformations which are associated with a predefined procedural pattern. On the other hand, WLS-to-WES contains the schema transformations that cannot be computed by the instance wrapper generator. They must therefore be programmed manually.

For simplicity, we assume that the following property of LPS-to-WES is respected:

$$\forall T_i \in \text{LPS-to-WES}, \ \forall T_j \in \text{LPS-to-WES},$$
$$((T_i \in \Gamma_{LW}) \wedge (T_j \notin \Gamma_{LW})) \Rightarrow \text{before}(T_i, T_j))$$

Both LPS-to-WLS and WLS-to-WES contain schema transformations that make implicit constructs and constraints. Referring to Figure 5-32, the implicit constructs are emulated by the wrapper through its query translation and data assembly components whereas the implicit constraints are handled by its integrity control manager.

We are now able to determine the definition components of the instance and upper wrappers:

- the wrapper schema (WES) for the query analysis;
- the structural mappings between the physical and wrapper logical schemas for the query translation: WES-to-WLS and WLS-to-LPS;
- the instance mappings that define the data assembly: lps-to-wls and wls-to-wes;
- the implicit constraints to simulate them: WES-to-WLS and WLS-to-LPS;
- the structure of the data source (LPS) for the access plan and optimization.

Wrapper language

Wrappers employ several languages and several technologies. 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 must use the legacy access techniques to improve data access performance. It therefore needs to access the native API of the underlying DMS by using the legacy language.

Wrapper generator algorithm

The wrapper generator algorithm is depicted in Figure 5-34. The objective of the wrapper is to generate the instance and upper wrapper components. After generating the query analyzer from the wrapper schema specification, the algorithm systematically analyses each schema transformation of LPS-to-WES. This analysis consists of two major steps. First, it generates the wrapper program code for each schema transformation $T_i \in \Gamma_{LW}$. Secondly, it inserts the definition of each transformation $T_j \notin \Gamma_{LW}$. In these two steps, the generated code depends on the type of the transformation expressed by the two functions: $type(T_i)$ and $IC(T_i)$. $type(T_i)$ returns the signature definition of the transformation whereas $IC(T_i)$ returns true if the schema transformation makes explicit an implicit constraint.

Moreover, for all the schema transformations T_i: $\neg IC(T_i)$, the algorithm updates an intermediate schema transformation sequence, denoted by S-to-LPS, that keeps all the transformations of WES-to-LPS that have not yet analyzed. This intermediate sequence enables to generate/define the implicit constraint assertions when they are not defined on constructs of LPS.

Finally, the algorithm generates the legacy access plan and primitives according to LPS and the underlying DMS (For COBOL, see Section 5.4.5).

```
Wrapper Schema:                    WES
Physical Schema:                   LPS
Schema Transformation Sequence:    <LPS-to-WES, lps-to-wes>


S ←  WES;
S-to-LPS ←  {Tj ∈ WES-to-LPS : ¬ IC(Ti)};

// Query analysis component (wrapper language transformation):

For each construct Ci ∈ WES do:
    generate wrapper query mapping(Ci);
end-for;

// Query translation/ result formation (schema mappings):

For each transformation Ti ∈ WES-to-LPS do:

    // Intermediate schema:
    S = Ti(S);

    // upper wrapper:
    if (type(Ti) ∉ ΓLW and ¬ IC(Ti))
    then
        generate transformation signature (Ti, S)
        S-to-LPS ←  S-to-LPS \ Ti;
    end-if;
    if (type(Ti) ∉ ΓLW and IC(Ti))
    then
        generate transformation signature (Ti, S, S-to-LPS)
    end-if;

    // instance wrapper generation:
    if (type(Ti) ∈ ΓLW and ¬ IC(Ti))
    then
        generate query and data mapping (Ti, S)
        S-to-LPS ←  S-to-LPS \ Ti;
    end-if;
    if (type(Ti) ∈ ΓLW and IC(Ti))
    then
        generate constraint assertion (Ti, S, S-to-LPS)
    end-if;
end-for;

// Access plan and optimization (legacy language mappings):

generate legacy access plan(LPS);
```

Figure 5-34: Algorithm of the instance wrapper generation.

5.5.3 Metrics

The costs of writing a wrapper can be expressed with respect to the number of its code lines[5]

(LOC). The total LOC is the sum of all the LOC of wrapper modules. For a particular DMS, a general formula for determining the total number of LOC can be specified as follows:

#LOC(Wrapper) =
#LOC(Model Wrapper) + #LOC(Instance Wrapper) + #LOC(Upper Wrapper)

The first component is a constant whereas the two other components depend on the wrapper schemas and the schema transformation sequence between them. The LOC cost of the instance wrapper depends on:

- the underlying DMS to which the wrapper is dedicated;
- the size of the wrapper and physical schemas;
- the number and the type of schema transformations of the sequence.

To illustrate this point, we have analyzed the LOC cost of our operational wrappers dedicated to COBOL files and relational data (Section 5.6). Figure 5-35 presents the average numbers of LOC needed by these wrappers for the emulation of certain constructs and constraints. The total size of the instance wrapper can be derived from these numbers.

Explicit Constructs	COBOL Wrapper	RDMS Wrapper
Entity type	380	125
Attribute	120	40

Implicit Constructs/Constraints	COBOL Wrapper	RDMS Wrapper
Implicit Compound Attribute	150	40
Implicit Multivalued Attribute	220	55
Implicit Identifier constraint	20	15
Implicit Reference constraint	30	15

Figure 5-35: LOC cost of explicit or implicit constructs and constraints.

To illustrate and compare the total number of LOC in the different wrappers, we have counted the total LOC of these wrappers on two quite different cases and studied how the size of the wrapper schema and the transformation sequence increase the number of LOC (Figure 5-36).

5. The number of LOC can be translated in other units (e.g., Euros).

	Wrapper Schema Size	Transformation Sequence Size	Implicit Constructs and Constraints
Case 1	3 entity types 15 attributes	3 transformations	2 implicit structures 3 implicit constraints
Case 2	30 entity types 120 attributes	60 transformations	35 implicit structures 35 implicit constraints

Figure 5-36: The two case studies and their size.

The results of the first and second case study are shown in Figure 5-37 and Figure 5-38 respectively.

Case 1	COBOL Wrapper	RDMS Wrapper
Model wrapper	4.345 (58 %)	7.458 (89,5 %)
Instance wrapper	3.173 (42 %)	865 (10,5 %)
Whole wrapper	7.518 (100 %)	8.323 (100 %)

Figure 5-37: Size of model and instance wrappers. Comparison between a COBOL wrapper and an RDMS wrapper for a wrapper schema made up of 3 entity types; 15 attributes; 1 implicit compound attribute; 1 multivalued attribute; 1 implicit identifiers and 2 implicit reference constraints.

Case 2	COBOL Wrapper	RDMS Wrapper
Model wrapper	4.345 (11 %)	7.458 (41 %)
Instance wrapper	33.675 (89 %)	10.650 (59 %)
Whole wrapper	38.020 (100 %)	10.108 (100 %)

Figure 5-38: Size of model and instance wrappers. Comparison between a COBOL wrapper and an RDMS wrapper for a realistic wrapper schema made up of 30 entity types; 120 attributes; 20 implicit compound attributes; 15 implicit compound and multivalued attribute; 35 implicit reference constraints and identifiers.

5.6 Operational Wrappers

To test our wrapper architecture, we have implemented wrappers for COBOL DMS and relational DBMS. The legacy data models for these DMS vary widely. Likewise, they provide query processing power that ranges from simple procedural mono-entity access to predicative multi-entity access.

As part of the InterDB project, we have implemented two kinds of wrapper: the InterDB log-

ical wrapper and the InterDB object wrapper:

- Whatever the underlying DMS, the *InterDB logical wrapper* provides a SQL-like interface to the underlying database. The InterDB logical wrapper is a software layer that offers (1) an abstract interface based on the wrapper logical model defined in Chapter 3; and (2) the *Logical Query Language (LQL)*, a variant of the SQL language which uses naming convention and terminology defined in the wrapper logical model.
- On top of the InterDB logical wrapper, the *InterDB object wrapper* provides an object-oriented view of the legacy database. Devoid of a query language, it gives a Java navigational object-oriented interface based on the wrapper object-oriented schema (WOS) of the underlying database.

InterDB wrappers and program applications communicate over the Internet using Java and the RMI communication protocol. With this architecture, a legacy database can be accessed in two ways (Figure 5-39): (1) through the logical wrapper interface by means of a query language or (2) through the object wrapper interface by means of methods.

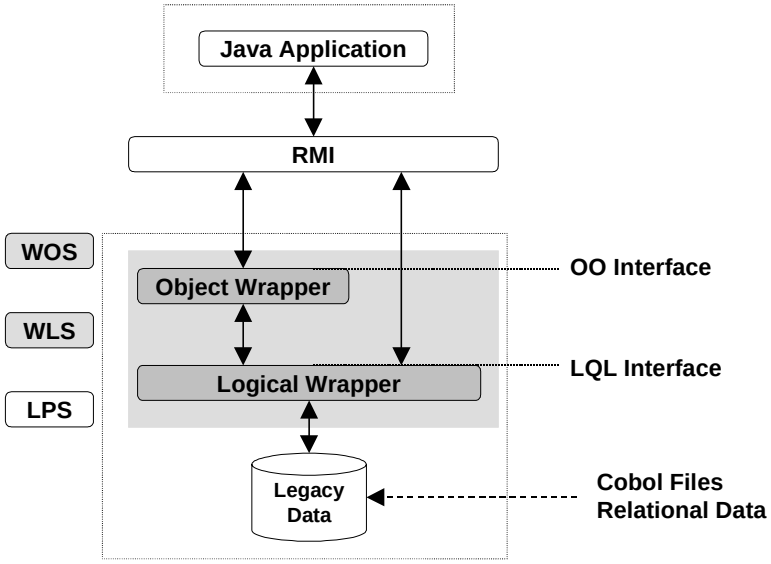


Figure 5-39: Logical and object wrappers in a Java environment.

Such an architecture provides an adequate way for solving the DMS and platform heterogeneity that appears when one wants to build a multidatabase or federated system. DMS independence is guaranteed by the logical wrapper. The wrapper provides the logical schema of the legacy database and a unique query language interface whatever the DMS wrapped. Platform independence is ensured by both the logical wrapper and Java.

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 networked applications 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 runtime system.

The *Java RMI* provides remote method invocations of 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).

5.6.1 InterDB Logical Wrapper

The InterDB 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 5-40). The logical module offers a unique interface to the applications whereas the logical middleware provides a transparent distribution across the network.

Logical middleware

The logical middleware is made up of the logical server and the InterDB driver, both dedicated to a database. The *logical server* manages the communications between the logical wrapper and the InterDB driver. It offers the object distribution using RMI as middleware[6]: it receives the queries sent by the InterDB driver and sends the result objects from the logical wrapper. The *InterDB driver* is a Java API for executing LQL statements and retrieving the result objects.

LQL statements. LQL is an SQL-like language modified to support the logical data model. LQL includes selection but also update queries. Analogous to SQL, LQL uses the select-from-where clause. LQL differs from SQL in that its select and where clauses can contain compound or multivalued attributes.

Java result objects. Result objects are constructed from entity types of the logical schema. The Java objects properties correspond to the attributes of the entity types. In the example schema of Figure 5-44, the entity types Employee and Office give rise to corresponding Java

6. Instead of an object-based middleware, we could use an XML-based middleware as well.

classes. Single-valuated simple attributes are modeled as simple properties (e.g. Integer, String or Date) whereas multivalued simple attributes are modeled as Vector. In our example, the attributes Reference and Name give rise to corresponding Java classes (respectively Integer and String). Compound attributes are defined as Java classes. This is the case for the Address attribute in our example. It is modeled as a Java class Address with two properties: Number and Floor. As an illustration, the Java definition of the entity type Office of Figure 5-44 is given in Figure 5-46.

More details about the Java API of the InterDB driver and the LQL query language can be found in [Thiran, 2002b].

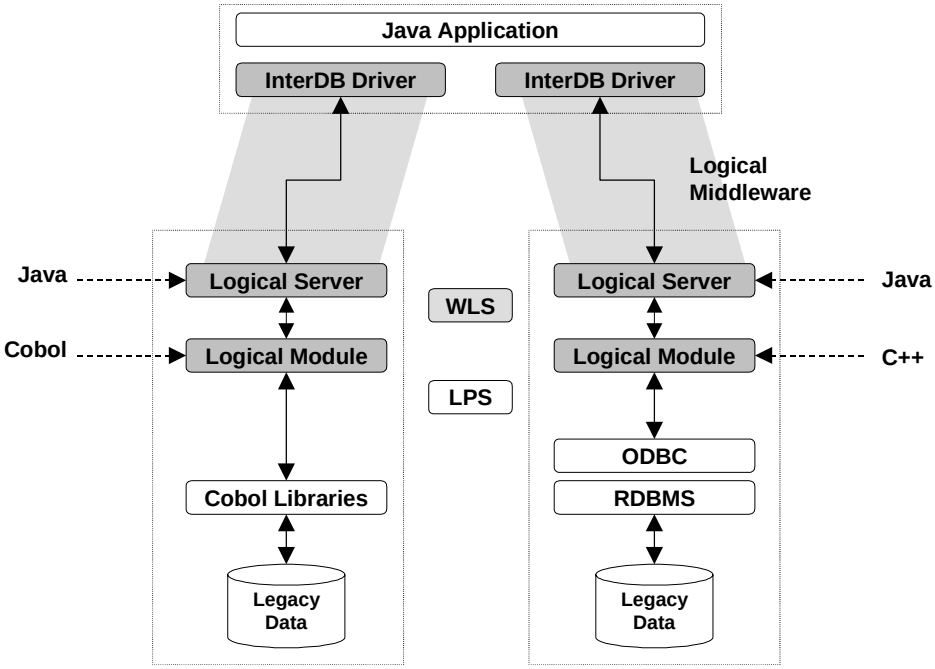


Figure 5-40: InterDB components of logical wrappers dedicated to COBOL files and relational databases.

Logical module

The logical module is in charge of managing the logical/physical conversion of the underlying legacy database. It 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 (Figure 5-41). In particular, it emulates implicit constructs such as foreign keys in COBOL files or multivalued fields in relational DB.

	Wrapper Emulation
Implicit Constructs	Compound attribute Multivalued attribute Optional attribute
Implicit Constraints	Identifier Reference

Figure 5-41: Implicit constructs and constraints that a logical module can make explicit.

For performance reason, we have decided to develop the logical wrappers as program components dedicated to a local database. In particular, the logical/physical mapping rules have been hardcoded in the modules rather than interpreted from mapping tables. Figure 5-42 depicts the set of schema transformations the logical modules can automatically emulate. Each of these schema transformations is associated with a predefined procedural pattern. The pattern instantiating is under the responsibility of *generators* that will be presented separately in Chapter 7.

	Schema Transformation
Constructs	Rename-ET Rename-Att Serial-CompAtt Single-CompAtt Serial-MultAtt Single-MultAtt Modify-Cardinality
Constraints	Create-Identifier Create-Reference

Figure 5-42: Schema transformations taken in charge by a logical module.

We have built hardcoded logical modules for relational databases (Oracle, InterBase and Access) and others for files managers (COBOL programs):

- *The modules for relational databases* have been written in C++ using ODBC for accessing the legacy databases. 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.
- *The modules for files managers* have been entirely written in COBOL. This implementation strategy has been motivated by the fact that the languages and the capacities of these DMS are different.

Figure 5-43 draws up the services the logical modules provide according to the underlying DMS. A yes denotes that the service is emulated by the logical module whereas a no denotes that the service is not taken in charge by the logical module. In this case, the service can be already provided by the underlying DMS. It mustn't be implemented by the wrapper.

Wrapper Service	Wrapper for Cobol Files	Wrapper for Relational DMS
Wrapper query analysis	yes	yes
Wrapper query mapping	yes	yes
Schema mappings	yes	yes
Legacy query mapping	yes	yes
Current concept	yes	no (delegated to ODBC)
Access optimization and processing	yes	no (delegated to DMS)
Error reporting	yes	yes
Semantic Integrity Control	yes	yes
Authorization control	yes	no (delegated to DMS)
Transaction management	no	no (but can be delegated to DMS)
Failure management	in part (memoryless wrapper)	in part (memoryless wrapper)

Figure 5-43: Wrapper services implemented by the InterDB prototypes, according to the underlying DMS.

Client application example

The client application example covers the basic use of the Java interface of logical wrappers. The Java client application illustrates the steps to connect, retrieve and update the results of a wrapper request. The application deletes all the offices that are in the fourth floor. The code includes a loop that accesses the offices of the fourth floor. In the loop, each result instance is associated with the delete statement. Any time something goes wrong between the JAVA application and the logical wrapper (for instance, if the implicit reference constraint defined on the entity type Employee is violated), then the former throws an exception.

Figure 5-45 shows the Java code of the first application whereas Figure 5-46 shows the Java definition of the entity type Office. Note that the LQL query expresses a selection on the compound attribute Address.

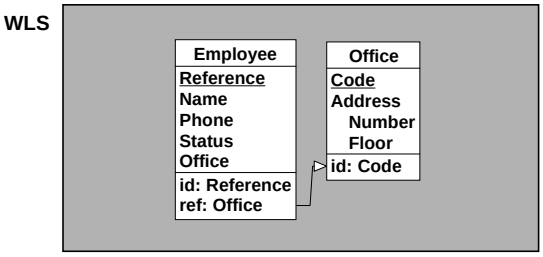


Figure 5-44: Wrapper logical example.

```
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();
    PreparedStatement ps = con.preparedStatement("DELETE OBJECT ? FROM OFFICE");
    EntityObject eo; // eo is the object that will represent the current instance of the ResultSet;
    ResultSet rs = stmt.executeQuery("SELECT * FROM OFFICE WHERE ADDRESS.FLOOR=4;");
    OFFICE off;
    if (rs.next())
                {
                  // get the values of the attribute of the object OFFICE:
                  off = (OFFICE)rs.getObject("*");
                  // displays the code of the current office:
                  System.err.println("Office.Code:"+off.Code);
                  // get the current instance of OFFICE for the update:

eo = rs.getEntityObject(); // eo represents the current instance of the ResultSet
                  ps.setObject(0, eo); //eo is bound to the delete statement
                  // execute the udapte statement:
                  if (ps.execute()) {,}
                }
  }
     catch (LQLException e) {System.err.println("LQL Error:"+e);}
  }
}
```

Figure 5-45: Java application example accessing the logical wrapper through the InterDB driver.

```
import java.io.*; // Serializable since these are distributed objects
import java.util.Vector;
public class Office implements Serializable {
        public Integer Code;
        public oaddress Address;
}
import java.io.*;
public class oaddress implements Serializable {
        public Integer Number;
        public Integer Floor;
}
```

Figure 5-46: Java definition of the object type Office.

5.6.2 InterDB 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 navigational object interface.

The object wrapper interface is made up of the objects defined in the wrapper object-oriented schema (WOS) of the underlying database (Chapter 2, Section 2.4.2) and a particular object, called the *access object*, that offers the sequential or direct access of any objects derived from WOS. 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 hides the LQL query language from the client applications. That is, it only provides an object-oriented framework (i.e., object types and methods) for manipulating read-only data. Methods are used to retrieve an attribute value of an object or to navigate between different objects. The navigation between objects derived from WOS is sequential (by means of methods getFirst*EntityObjectName*, getNext*EntityObjectName* or get*EntityObject-Name* methods). However, any object derived from WOS can be directly reached by means of its identifier by means of a method get*ObjectEntity* provided by the access object.

The object wrapper makes possible to modify objects and to readily define user-defined objects that can represent user views of 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, 1988].

The details of the object wrapper implementation can be found in [Thiran, 2002c].

Client application example

To illustrate the remote object interface of the object wrapper, we take up the wrapper object schema of Figure 5-47. Each entity type corresponds to a Java remote object that can be manipulated by any Java client applications through the RMI protocol. An example of the Java remote object definition of Office is shown in Figure 5-48 and an example of a Java client ap-

plication is given in Figure 5-49.

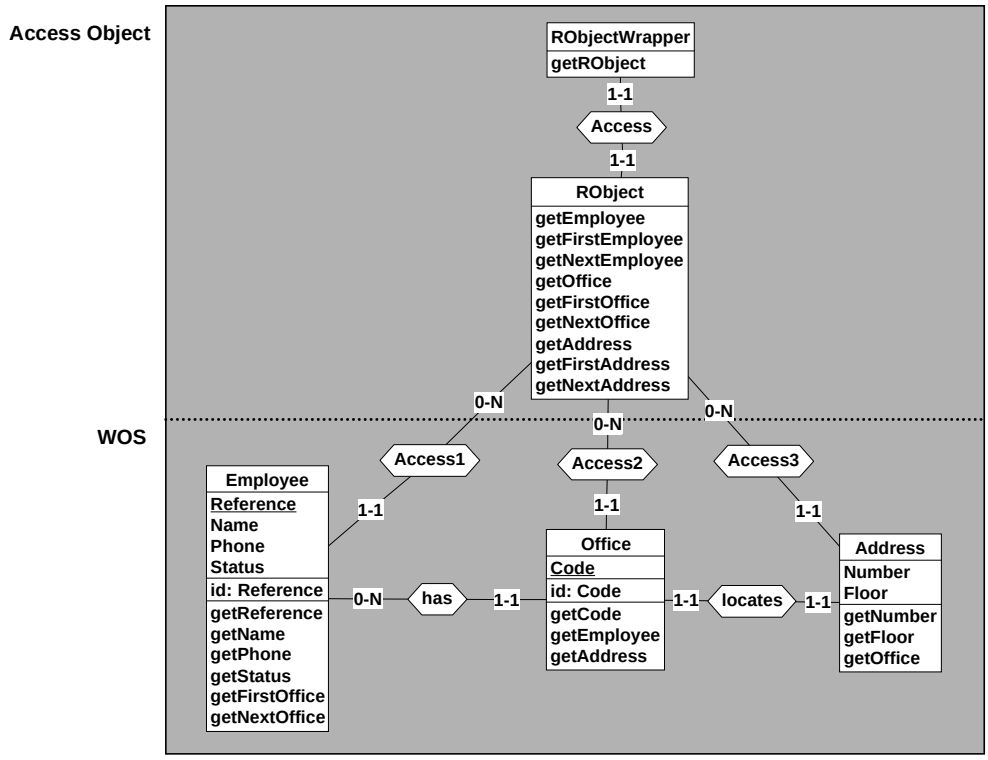


Figure 5-47: Wrapper object interface example: the object types derived from WOS (bottom part) and the access object that provides sequential and direct access to any explicit objects of WOS.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Office extends Remote
{
Employee getEmployee() throws RemoteException, DOException;
Integer getCode() throws RemoteException, DOException;
Address getAddress() throws RemoteException, DOException;
}//End interface
```

Figure 5-48: Remote object interface example: the object type Office.

```
import java.rmi.*;
class Client
{
    public static void main(String[] args)
    {
        RobjectWrapper remote;                  // remote server interface
        RObject ro=null;                          // remote access object interface
        try
        {
            //Remote interfaces of classes
            Office off;
            Address add;
            //Client connection:
            remote = ObjectServer.getConnection(maverick.info.fundp.ac.be:1099/Server);
            //navigating through objects:
            ro = remote.getRObject();
            off = ro.getFirstOffice();
            while (off != null)
            {
                System.out.println( "Code: " + off.getCode());
                add = off.getAddress();
                System.out.println("Address.Number: " + add.getNumber());
                off = ro.getNextOffice();
            }
            ro.close(); remote.close();
        }
        catch (BOException e)
        {System.out.println("BOException: " + e.getMessage());}
    }
}
```

Figure 5-49: Java application example accessing the object-oriented wrapper.

Part III

Methodology

Chapter 6

Forward-Reverse Methodology

In which the combined forward-reverse methodology of the schema-oriented framework for data mediation is presented. The forward and federation engineering approaches are combined with the central idea that they are tightly bound. In that way, we state that the global schema is defined not only by the actual requirements but also by the description of the database federation.

6.1 Introduction

In Chapter 4, we have presented the schema-oriented framework of data mediation. In this framework, the mappings defined as schema transformations are the unifying links between the architecture components and their dedicated methods. In this chapter, we develop the methods that are to help developers define the schema hierarchy and the mappings.

One key feature of the schema-oriented framework is that the global schema is defined not only by the contribution of the legacy databases but also includes the actual requirements. Hence, the need for a methodology that offers a virtual and integrated view of the legacy resources and the new requirements. We are faced with two distinct engineering domains (Figure 6-1):

- the *forward engineering*: designing a global/federated schema that models the new requirements;
- the *backward engineering*: developing a federated schema that offers an integrated view of the underlying legacy databases.

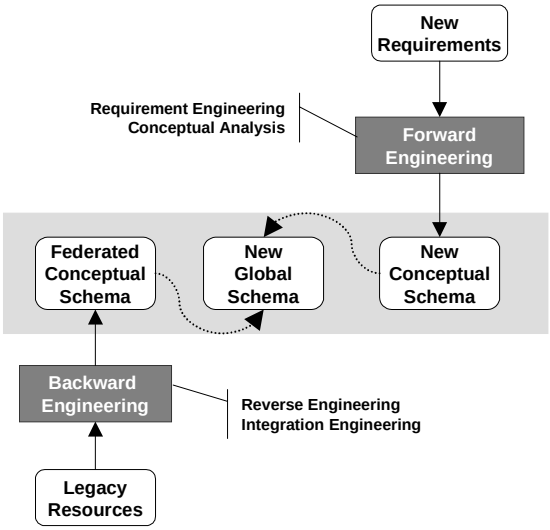


Figure 6-1: Forward and backward methodologies.

Most previous approaches ([Sheth, 1990], [Parent, 1998], [Hainaut, 1999]) propose a backward approach in which the global schema is defined from legacy information only. There are only a few attempts that consider the importance of the acquisition of the actual requirements to interrelate legacy information.

We consider here both of engineering approaches and combine them with the central idea that they are tightly bound. In this chapter, we describe a general methodology based on a conceptual data description and intended to find out which part of the actual requirements can be met by the legacy systems and which part has to be managed by additional data systems.

The rest of this chapter is organized as follows. Section 6.2 examines the backward integration methodology. The objective is to provide a general framework for database integration. Section 6.3 discusses the main baselines of the forward-reverse methodology. We outline key ideas on which our approach to the forward-backward integration is based and specifies the specific sub-problems on which our approach focuses. The last sections of this chapter develop each step of the methodology.

6.2 Backward Methodology

The purpose of this section is to provide a clear picture of what the main issues and the current solutions in the schema integration field are. The focus is on the concepts, the proposed solu-

tions and not on detailed technical discussions. We illustrate the backward methodologies by presenting a practical approach we have developed as part of DB-MAIN ([Thiran, 1998], [Hainaut, 1999]).

6.2.1 Baselines

This section provides a survey of the most significant trends in backward methodologies. Backward methodologies consider the global schema as the result of *database integration*. Simply stated, database integration is the process which takes as input a set of (existing) databases and produces as output a single unified description (the global/federated 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], [Hainaut, 1999], [Sattler, 2003]), primarily because the same fact can be contained in several databases yet be represented using different conceptual structures.

To provide a framework for schema integration, we first outline the main steps of typical methodologies and present the main issues of schema integration. Due to the inherent complexity of this task, schema integration cannot be performed in an ad hoc fashion. Usually, the process of schema integration consists of several steps. [Batini, 1986] and [Spaccapietra, 2000] identify the following phases (Figure 6-2):

- *Schema preparation* where local schemas are transformed to make them homogeneous (both syntactically and semantically);
- *Schema comparison* devoted to the identification and categorization of interschema relationships;
- *Schema conformation* which solves interschema conflicts;
- *Schema merging* which merges schemas into one global one;
- *Mapping definition* which involves storing information about the inter-schema mappings.

Schema integration requires the definition of relationships between schema elements of the local schemas to be integrated. The detection and definition of such relationships is complicated by the semantic heterogeneity of the components [Garcia, 1996]. A semantic taxonomy ([Sheth, 1993], [Kashyap, 1993]) containing terms like *semantic proximity* or *semantic incompatibility* is necessary for semantic reconciliation ([Sheth, 1991], [Sheth, 1993]). It has been widely agreed [Sheth, 1990] that schema integration cannot be completely automated, as this would require that all of the semantics of the local schemas would be completely specified. It should be noted, that, in general, semantic conflicts can only be identified by human reasoning based on domain and common sense knowledge [Navathe, 1996]. Nevertheless, the user needs tool support to perform this complex task (Chapter 7).

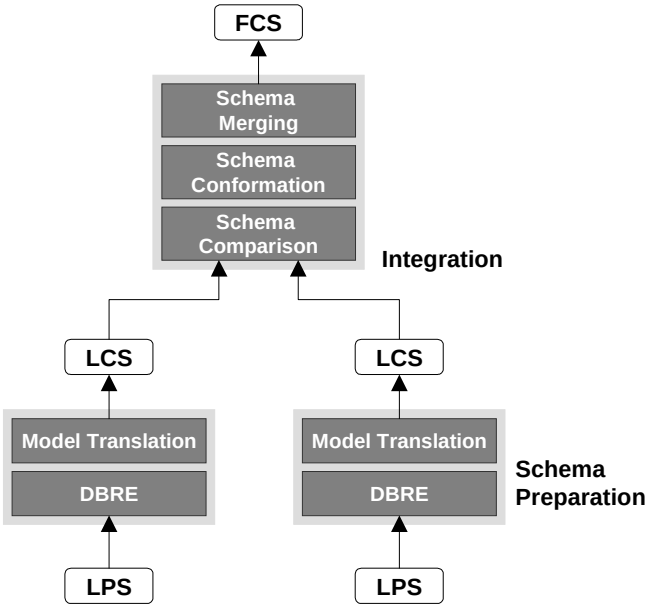


Figure 6-2: Framework for schema integration: the main steps.

In this section, we take up and develop the example that has been used in Chapter 4. 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 6-3). Through this example, we have already illustrated some of the problems of the schema preparation phase (i.e. semantic enrichment and syntactic rewriting). This example is the starting point for the illustration of the integration processes.

Schema preparation

In this phase, schemas that correspond to the individual databases being integrated are translated into schemas using a canonical data model. This phase partially corresponds to the DBRE and model translation processes presented in Chapter 4:

- *Semantic enrichment (DBRE).* 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 analyses makes it necessary to recover hidden structures and constraints (Section 6.4 and Chapter 4, Section 4.2.3).

- *Model translation.* Local schemas are translated into a canonical data model (Chapter 4, Section 4.2.4). This allows for resolving syntactic heterogeneity that is the result of different data models.

Example

Figure 6-3 shows the schemas extracted from the relational database and of the COBOL files according to their data model.

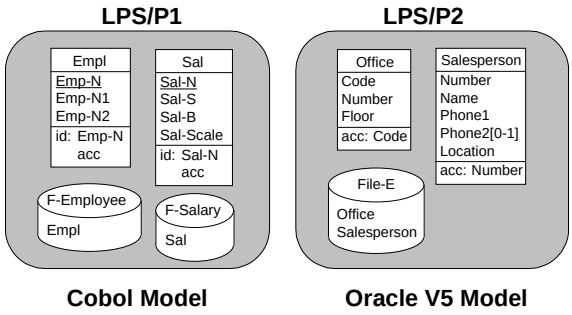


Figure 6-3: The local physical schemas of the COBOL files (left) and of the relational database (right). LPS/P1 comprises two files and two record types (Empl and Sal). Sal-N and Emp-N are unique record key. LPS/P2 comprises two table, namely Salesperson and Office.

The physical schemas of Figure 6-3 have refined through in-depth inspection of the way in which the COBOL programs and SQL DML statements use and manage the data in order to detect the implicit constraints and constructs (Chapter 4, Section 4.2.3). Moreover, names have been reworked and physical constructs are discarded. Finally, the resulting schemas are expressed in a variant of the Entity-relationship model (Figure 6-4).

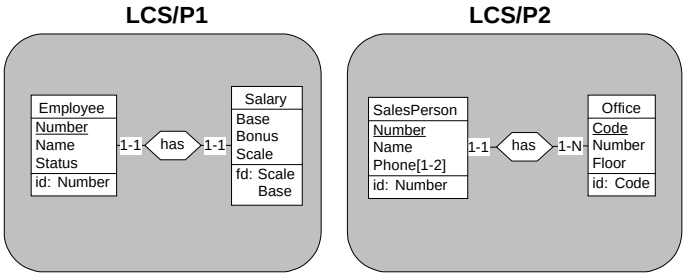


Figure 6-4: The local conceptual schemas (LCS) of the COBOL files (left) and of the relational database (right) resulting from the schema refinement described in Chapter 4, Section 4.2.3.

Schema comparison

Each local schema corresponds to an intension and an extension. The *intension* of a database schema is determined by a set of structural definitions. The *extension* of a schema refers to an actual database state at a given time. A correct and complete schema comparison has to deal with both parts.

At the intensional (or structural) level, the objective is to identify constructs in the underlying schemas that can 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. Their semantics 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 classification of the relationship among the constructs 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 from 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.

At the extensional (or instance) level, the objective is to determine instances of constructs in different sources that refer to the same real-world entity. The simplest approach assumes that types from different databases have a common identifier. Hence, types that have a common identifier value identify the same real-world entity [DeMichiel, 1989]. However, a common identifier is not always available. This is referred to as the *identifier equivalence problem*. More details can be found in [Savasere, 1991] and[Elmagarmid, 1999] and [Ramesh, 1997].

To support the interschema correspondence identification, sophisticated dictionaries, thesauri or *ontologies* can be used. An ontology is a repository of all current knowledge in the organization or beyond [Lee, 1996]. The ontology describes the semantics of all concepts and of their relationships between, and is therefore capable of correctly identifying interschema correspondences [Parent, 1998]. An example of such an ontology is WordNet [Miller, 1993]. It can serve as a common ontological framework [van den Heuvel, 2002b]. This web-based ontology focuses on meanings of terms rather than forms, and, incorporates a taxonomy comprising synonym sets, hyponymy/hypernymy, and the like.

Example

In Figure 6-4, we establish that the entity type Salesperson of P2 and the entity type Employee of P1 are related to each other. Moreover, we identify that the attributes Number and Name in the two entity types are related. Finally, data analysis, i.e. examination

of actual instances of the physical data types shows that the employees (Salesperson and Employee values) that work on both sites are represented in both databases.

Schema conformation and merging

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 can exist between related constructs.

Taxonomies of conflicts abound in the literature, from very detailed ones [Sheth, 1992] to simpler ones [Spaccapietra, 1991]. In [Thiran, 1998], we have classified the conflicts according to three possible categories: 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. For instance, an OrderDetail instance can be represented by an entity, by an attribute value and by a relationship.
- *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 in B, A and B 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.* They 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 inst(Employee) $\supseteq$ inst(User), where inst(A) denotes the set of instances of data type A at a given time. This problem has been discussed in [Vermeer, 1996]. This process is highly knowledge-based and cannot be performed automatically.

Example

In the integrated schema, the entity types Employee and Salesperson have been merged into a common entity type. Note that the semantic conflicts among the attributes and roles have been resolved by relaxing the constraints.

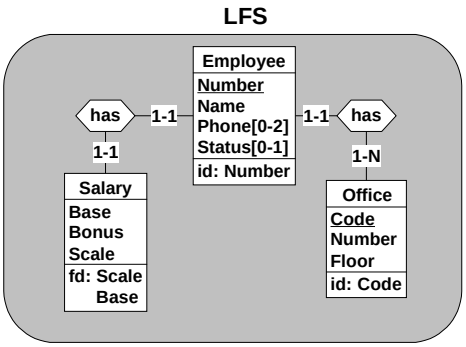


Figure 6-5: Global conceptual schema.

6.2.2 Practical DB-MAIN Methodologies

As part of the DB-MAIN project [Hainaut, 1999], we have developed a practical approach of the database integration. This approach proposes a comprehensive process mainly based on a structural analysis of the database. Experience has shown that this process must be coped with through very flexible techniques, and that different problems in the same federation can require different techniques. Two main complementary strategies have been proposed. They are 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.

These two strategies are supported by the DB-MAIN CASE tool (Chapter 7, Section 7.4.5).

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. *Schema preparation.* 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 ET_2 in schema 2 is recognized as a subtype of ET_1 in schema 1, then an empty entity type with name ET_1 is created in schema 2, and made a supertype of ET_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.

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).

6.2.3 Discussion

The backward strategy of integration has fundamental implications on the global schema definition, on its flexibility and evolvability. In this section, we carefully analyze the backward strategy according to these points. We do not intend to be exhaustive here, rather, we select some relevant basic assumptions that limit the applicability of such a strategy.

Global schema definition

The backward strategy builds the global schema from the legacy information resources towards the global schema, using a database integration method like that presented in the previous section. This results in a global schema that enjoys the following properties:

- It directly depends on the legacy local schemas and the integration method used.
- It does not integrate the requirements of the new applications that are to be developed on top of the legacy databases.
- It often holds more information than required for the actual requirements of the organizations. Since the relevant information is hidden in a global schema, the user is responsible for finding the required information.

The most serious limitation of the classical integration approach is that it requires too much and too broad global knowledge to generate the global schema [Aslan, 1999]. It also requires a large global structure to be maintained. The required integration effort is very high.

Example

Figure 6-6 illustrates the differences between the integration process based on the backward approach described previously (Section 6.2.2) and the integration process that uses combined forward-backward processes (Section 6.6). The differences between the resulting schemas can easily be observed in this figure: the backward approach produces a global schema more complex than that produced by the forward-backward approach.

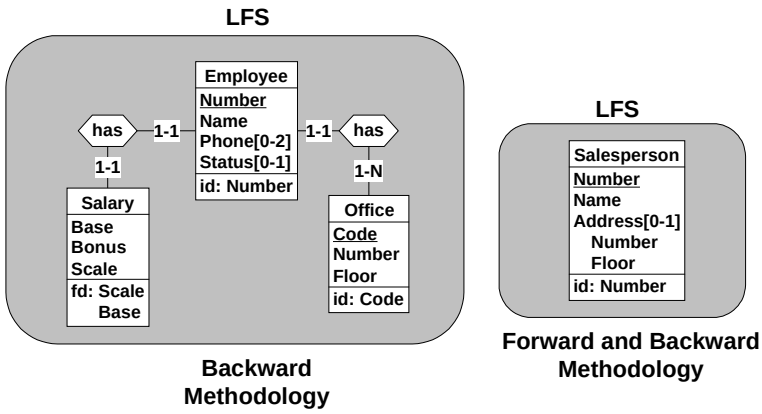


Figure 6-6: Integrated schema comparison. The left schema is the result of the classical integration approach whereas the right schema is the result of the combined forward and backward methodology.

Flexibility and evolvability

The process of integration greatly affects the *flexibility* and *evolvability* of the global schema. A change in the configuration or in one legacy database triggers a new integration process, resulting in an adapted global schema. As a result, the schema integration is not appropriate in scenarios where sources are often quickly removed or new sources are added or if the global requirements themselves are changing. The complexity and the result of the process integration depends on the number of registered sources.

6.3 Forward-Reverse Methodology Principles

The new global schema (NGS) is the answer to the requirements of the organization as they are perceived from now on. As far as information is concerned, it represents the services that the new system will be able to provide. Since the approach is based on reusing existing resources as far as possible, the future system will comprise legacy databases as well as a new one, so that the requirements will be met by both kinds of databases. Therefore, one of the

challenging problems is the precise distribution of the requirements among these components. We propose to resolve the integration by combining forward and reverse processes. Unlike [van den Heuvel, 2002], we believe that reverse and forward processes are tightly bound.

6.3.1 General Architecture

The general architecture of the methodology is outlined in Figure 6-7. The main methods are: (1) preparation (DBRE and model translation); (2) homogenization; (3) legacy-legacy integration; (4) global-legacy comparison.

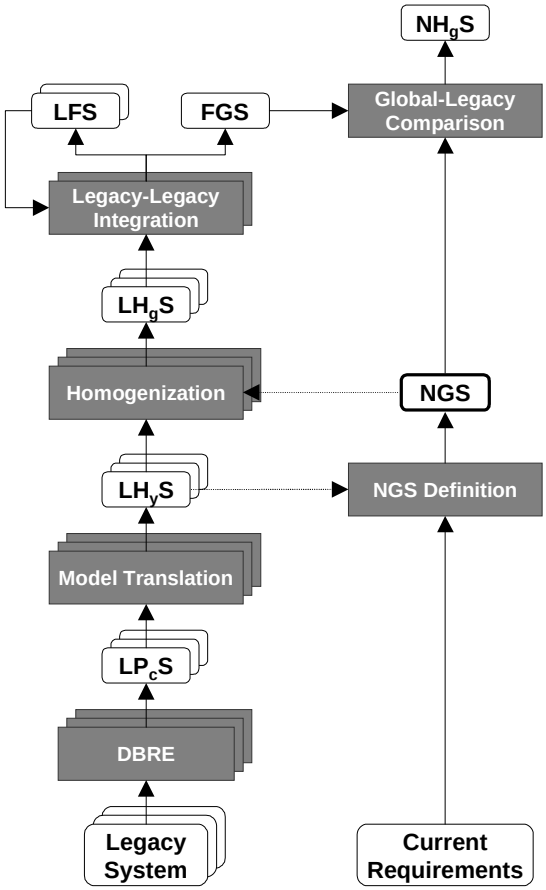


Figure 6-7: The main processes of the methodology.

All these methods are part of the schema-oriented framework defined in Chapter 4. Mappings defined as schema transformations are the unifying element between the methods and the architecture components: a method defines the schema transformation sequence that is used to automate the translation of queries within the corresponding architecture component.

We first outline the key ideas on which our approach is based and specify the problems on which our approach focuses. We then describe the methods of the methodology and the mappings they intend to define. We illustrate how these methods deal with the hybrid system described in Chapter 1, the common case study used throughout this thesis.

6.3.2 Integration *revisited*

We propose here an integration methodology which considers both the legacy resources and the new requirements. In our approach, each data source in the access scope of NGS is first homogenized in regard to NGS and then integrated into NGS. Once homogenized, a database source can be integrated into NGS through a simple integration mechanism using the synthetic strategy presented in Section 6.2.2. The methodology is heavy in schema preparation (DBRE and homogenization) but light in integration. Homogenization counts for most of the effort for including a new legacy source. However, multiple sources can be homogenized in parallel: homogenization involves only a legacy database source and can be managed by local owners and maintainers (e.g., modelers, administrators, current users and application maintainers) who are the best experts on the semantics of information sources they maintain in their databases. This results in a methodology that splits up and distributes the integration tasks and responsibilities (Figure 6-8).

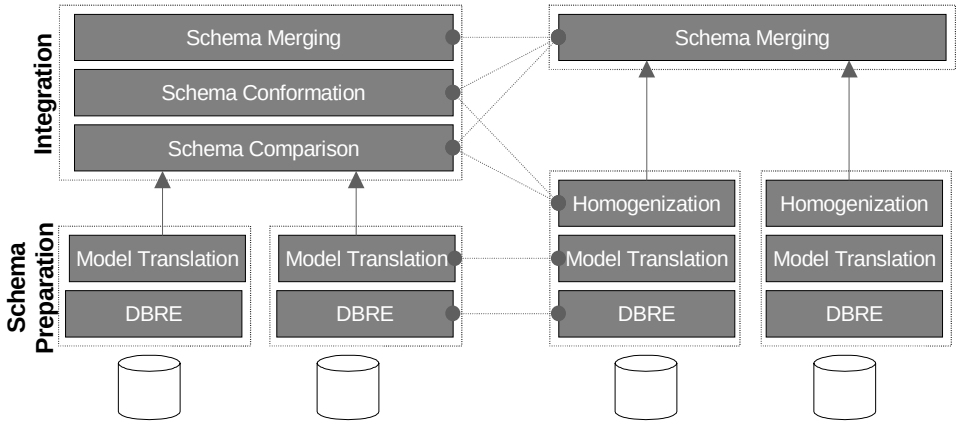


Figure 6-8: Classical database integration methodology vs database integration methodology within the forward-reverse methodology. The schema preparation phase in the forward-reverse methodology includes the DBRE, model translation and homogenization. The homogenization includes schema comparison and conformation of each local schema against NGS.

The backward and forward methodology proposes an approach that should bring some important advantages, such as: scalability, progressivity, legacy preservation, maximal reuse and simplicity of the integration process.

- *Scalability:* new levels and new legacy databases can be incorporated with minimal cost. It is designed to facilitate dynamic and scalable data integration. Its complexity is independent of the number of sources registered. Including a new database in the federation involves modification that mainly is taken in charge by the local sources.
- *Progressivity:* each step provides the formal definition of a layer of the schema-oriented framework that can be exploited immediately; this also ensures better development cost and time control, as well as better risk management.
- *Legacy preservation:* local databases and applications can be used while their functions meet the local needs.
- *Maximal reuse:* since the semantic structures of the legacy databases have been precisely elicited, the exported schema of new databases to develop includes the new requirements, and only them.
- *Simplicity of the integration process*: it requires minimum expertise to manage. Issues generally addressed in theoretical approaches to schema integration are of a lesser importance in our framework since they have been performed in the reverse engineering and homogenization processes. This is the case for conflict identification and conflict resolution. Indeed, the reverse-engineering process has given analysts a strong knowledge of the semantics of each construct of the local conceptual schemas. In addition, the homogenization step has produced fairly homogenized schemas in which few discrepancies 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.

6.3.3 Mapping Definition

We distinguish the vertical and horizontal correspondences to define the relationships between schemas (Figure 6-9):

- *Vertical correspondences* specify the relationships between two schemas of different layers of the schema hierarchy.
- *Horizontal correspondences* state the relationships between two schemas of the same layer of the schema hierarchy.

The vertical correspondences are formalized as a chain of transformations whereas the horizontal correspondences are defined by means of interschema correspondence assertions.

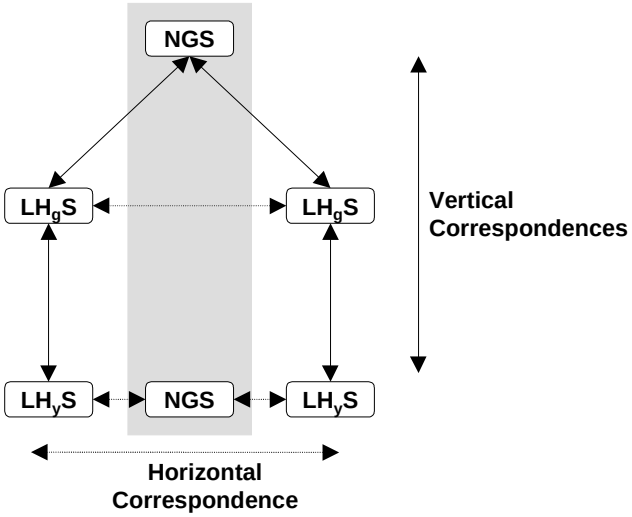


Figure 6-9: Vertical and horizontal correspondences.

Horizontal correspondences

As mentioned before, schema integration consists in determining the interschema correspondence assertions (ICA) by considering both the intensional and extensional levels. At the intentional level, the objective is to identify constructs that are semantically related and to classify the relationships among these constructs. An example of such relationships have been proposed in the InterDB methodology (Section 6.2.2). All conflicts detected in this step have to be resolved.

At the instance level, the constructs of the local schemas are usually set into a relationship with respect to their extensional assertions [Spaccapietra, 1992]. According to [Spaccapietra, 1991], the following binary extensional assertions can be specified between four classes:

- *disjointness* ($\neq$): the disjointness assertion states that the two types are extensionally disjoint in each corresponding database state.
- *equivalence* ($\equiv$): the equivalence assertion says that the two types are always extensionally equivalent.
- *containment* ($\supseteq$): the containment assertion is used to describe the fact that one type always extensionally contains the other type.
- *overlap* ($\cap$): the overlap assertions means that the two types can overlap, that is, they may contain instances that refer the same real-world objects.

Vertical correspondences

For any stage of the schema hierarchy, a set of schema transformation rules can be defined using the generic transformation set described in Chapter 3. The types of schema transformations that should be applied depend on the kind of horizontal correspondences to be set or conflicts to be resolved. It should be noted that as far as our notion of schema transformation is concerned, there is really no distinction among the steps of the methodology: we use the transformational paradigm to define mappings between any pair of schemas.

Horizontal and vertical correspondences

The integrated schema is obtained by a set of schema integration operations. The following list enumerates the basic transformational operations for «integrating» two entity types ET_1 and ET_2 with the respective intensions I_1 and I_2 and extensions C_1 and C_2.

- *Generalization* ($\vee$): 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* ($\wedge$): 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[1] of the extensions of the entity types ET_1 and ET_2.
- *Subtype* ($\Uparrow$): one entity type becomes the subtype (supertype) of the other entity type in the global schema.
- *Merging* ($\oplus$): 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 (Chapter 3, Section 3.5.3).
- *Partitioning* ($\otimes$): 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 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. Analo-

1. The extension of the new entity type is included in the intersection of the extentions.

gously, 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.

- *Linking* ($\Leftrightarrow$): the entity type ET_1 and ET_2 are linked by an entity relationship RT.
- *Preservation* ($\perp$): the entity types ET_1 and ET_2 are taken over into the global schema without any intensional and extensional modifications.

The basic schema integration operations are illustrated in Figure 6-10.

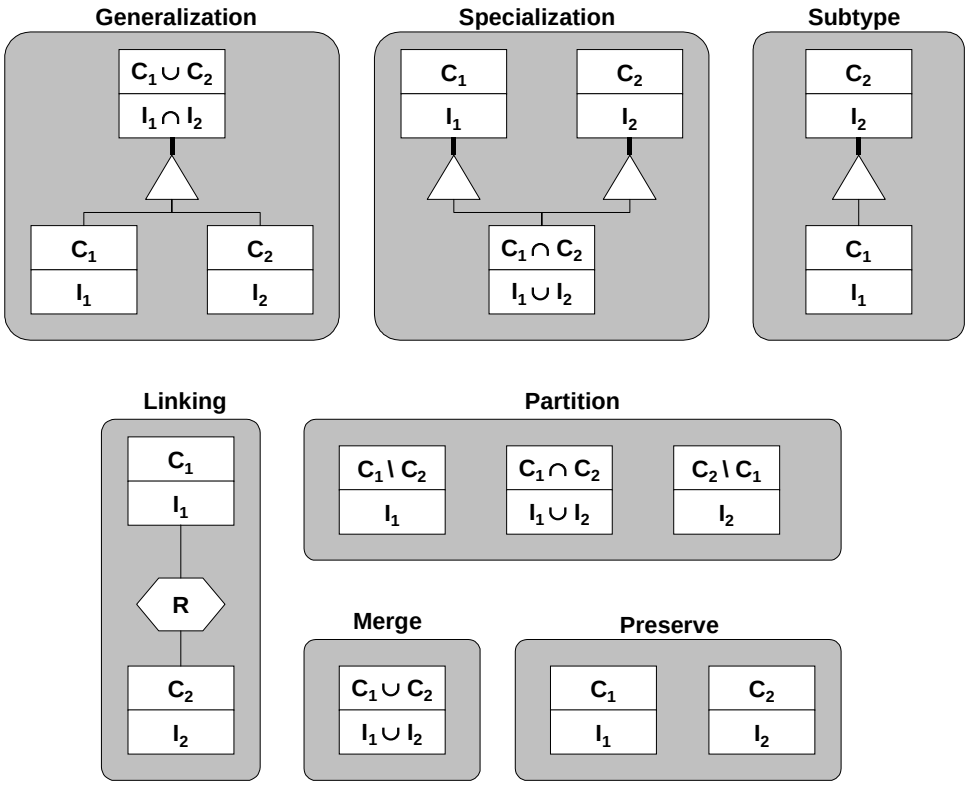


Figure 6-10: Schema integration strategies on entity types.

Figure 6-11 shows the applicability of the schema transformation above in case of different vertical correspondences [Dupont, 1996]. A + denotes that the operation is particularly well suited whereas a - denotes that the operations is impossible or incorrect. The open fields refer to a combination which is indeed possible but usually not-suited. The merge operator is applicable in each case. The input entity types are merged into one class. Depending on the ex-

tensional assertion, the schema integration is performed. For instance, in case of the disjointness assertion, the instances of the entity types are simply merged into a common extension. This can be done by the generalization and merge operators.

	$\vee$	$\wedge$	$\Uparrow$	$\oplus$	$\otimes$	$\Leftrightarrow$	$\perp$
$\neq$	+	-	-	+	-		
$\equiv$		+		+	-		
$\supseteq$			+	+	-	+	
$\cap$	+	+	-	+		+	

Figure 6-11: Compatibility of extensional assertions and transformation operations (from [Dupont, 1996]).

6.3.4 User-defined Function

In Chapter 3, we have introduced a special kind of schema transformations: Create-Function-Group. This transformation type defines that the components of the added group are calculated by using a *user-defined function* F whose semantics is known by the user only. It has no predefined t mapping part, so that the instance conversion cannot be predefined, but must be manually written. We consider two main classes of user-defined function:

- *Conversion function* and *mapping table* for resolving conflicts due to differences in schema definition;
- *Reconciliation function* for resolving conflicts due to inconsistency in data values.

Conversion function

For resolving domain and semantic discrepancies (Section 6.5.3), *conversion functions* are introduced. Such functions are used to transform attribute values. 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.

We define the *conversion function* as follows. Let us consider an attribute A that is a component of a group of an entity type ET, Fct: $t_A \rightarrow t$ is a function which can be applied to values of domain t_A defined for the attribute A of ET resulting in values of domain t.

Mapping table

Another way of transforming attribute values is the usage of *mapping tables*. A table Mapping is associated with an attribute group if the instance mapping definition of its component is expressed as Mapping(source, target, default) where source and target are attributes of the mapping table Mapping describing the instance mapping and default is an optional default value which is used if no explicit instance mapping is provided for some attribute values.

From an operational point of view, the transformation of local values works as follows: taking a value of the attribute of the group, we look for an instance in Mapping having this value as value in the attribute source. If such an instance exists, its attribute target contains the transformed value; otherwise the result is the default value if given.

Reconciliation function

For instance conflicts (Section 6.6.3), a reconciliation function is used for resolving inconsistencies among certain data of different database sources. A reconciliation function is a *user-defined function* which is called for each instance fulfilling a comparison condition (like the union). The affected instances are passed as arguments to the function, the resulting instance leads to a conflict-free result. In this way, the value of any attribute can be computed from the (possibly) conflicting values of other corresponding attributes.

An example of using a reconciliation function as part of the merging operation ($\oplus$) is the situation where two local databases contain partially the same salespersons. If the corresponding instances contain different values for the same attribute, the reconciliation function is able to select one of the conflicting values for the resulting instance or to compute a new one.

Illustration

This section presents the example-driven principle of data merging and conflict resolution. The main idea is to present the main applications and mechanisms of the user-defined functions. Figure 6-12 depicts a simple example of merging operation. The global entity type Product is defined as the union of the intension and extension of two local entity types Product of S_1 and S_2.

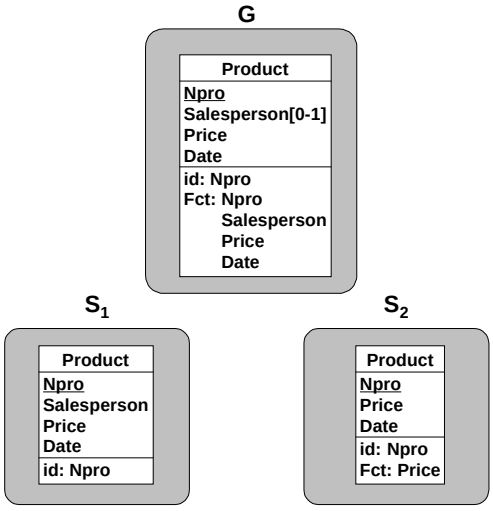


Figure 6-12: Example of an entity type resulting of a merging strategy.

The two entity types exploit some typical conflicts:

- *Domain discrepancy:* in S_1, the price data are recorded in Euro while in S_2, the same data are recorded in Belgian Franc.
- *Missing attribute:* in S_1, the entity type includes the attribute Salesperson that is missing in S_2.
- *Data inconsistency:* for a same product number (Npro), the prices can be different according to the sources.

These conflicts are resolved by means of two user-defined functions:

- *Currency conversion:* a function group (BEFtoEuro) made up of the attribute Price is defined in the schema S_2 (Figure 6-13).
- *Reconciliation function:* the global entity type includes a function group made up of the conflicting attributes. That is, we assume that the price in G is the most up-to-date price. Therefore, if the product entity of S_1 refers to an earlier product, it should be used, otherwise the more recently price value from the corresponding product entity in S_2 appears in the integrated result (Figure 6-14).

```
Numeric BEFtoEuro(Numeric PriceBEF)
{
    Numeric PriceEuro
    PriceEuro ← PriceBEF / 40.3399;
    return PriceEuro;
}
```

Figure 6-13: Conversion algorithm.

```
Product resolveConflict(Product P1, Product P2)
{
    // P1 is an instance of a product of S1, P2 is an instance of a product of S2
    // P1 and P2 have the same Npro value (matching attribute)
    Product G;
    G.Npro ← P1.Npro; // copy npro
    G.Salesperson ← P1.Salesperson; // copy salesperson
    // Resolve Price attribute:
    if (P1.Date > P2.Date)
          then {G.Price ← P1.Price;
                G.Date ← P1.Date;}
          else {G.Price ← P2.Price;
                G.Date ← P2.Date;}
    return G;
}
```

Figure 6-14: Reconciliation algorithm.

For understanding the user-defined function application, we assume a simple query posed on the global schema G (Figure 6-15). By applying the transformation rules presented in Chapter 3, we obtain a global query plan made up of the local queries posed on the two local schemas S_1 and S_2. The query plan uses the matching operators (union) and the reconciliation function to build the instances of the global result set using the corresponding local instances based on the common identifier values.

Query on G
[**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**att**, *Product, Salesperson,* PR, S], [**att**, *Product, Date,* PR, D]]
Retrieve the number, the price, the salesperson and the sale date of all the products

Query on S_1	**Query on** S_2
[**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**att**, *Product, Salesperson,* PR, S], [**att**, *Product, Date,* PR, D]]	[**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**void**], [**att**, *Product, Date,* PR, D]]

Query on G	**Query Plan**
[**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**att**, *Product, Date,* PR, D], [**att**, *Product, Salesperson,* PR, S]]	[**and**, [**plan**, **union**, *ResolveConflict* [**source**, S_1, [**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**att**, *Product, Date,* PR, D]]] [**source**, S_2, [**and**, [**att**, *Product, Number,* PR, N], [**att**, *Product, Price,* PR, P], [**att**, *Product, Date,* PR, D]]]], [**source** S_1, [**att**, *Product, Salesperson,* PR, S]]]

Figure 6-15: Global query decomposition and global query plan.

To illustrate the instance mapping principle, we use the example of Figure 6-15. Assume that the S_1 and S_2 represent instance sets as shown in Figure 6-16. By applying the currency conversion function and the merging operator between S_1 and S_2, the query plan derives instance set as shown in Figure 6-17.

S_1.Product			
Npro	**Salesperson**	**Price (Euro)**	**Date**
1	Michaux	123	1/01/2002
2	Gide	10	1/05/2002

S_2.Product		
Npro	**Price (BEF)**	**Date**
1	4420	1/03/2002
2	400	1/01/2002
3	4000	1/06/2002

Figure 6-16: Instance sets of $S_1$and S_2 (S_1.Product and S_2.Product).

G.Product			
Npro	**Vendor**	**Price (Euro)**	**Date**
1	Michaux	123	1/01/2002
1	null	109.58	1/03/2002
2	Gide	10	1/05/2002
2	null	9.92	1/01/2002
3	null	99.16	1/06/2002

Figure 6-17: Derived instance set of the global entity type (G.Product).

In Figure 6-16, S_1 indicates that product 1 is sold at 123 Euros while S_2 indicates that the same product is sold at 109.58 Euros. This conflict is reflected in Figure 6-17 as a violation of the identifier since they are more than one instance with Npro=1. This indicates a data inconsistency.

For example, we have the following set of instances wrt Npro in Figure 6-17:

Npro=1 : {(1, Michaux, 123), (1, null, 109.58)}
Npro=2 : {(2, Gide, 10), (2, null, 9.92)}
Npro=3 : {(3, null, 99.16)}

As a result, there are conflicts with Npro=1 and Npro=2.

At this point of the instance mapping, any instance conflicts has not yet removed. The conflict resolution is performed by applying the reconciliation function resolveConflict. More precisely, the reconciliation function resolves the conflict values for all the instances of the derived instance set that have a common Npro value.

The reconciliation function resolveConflict is applied for Npro=1 and Npro=2. That is, the function resolveConflict (Figure 6-14) resolves conflicts on attribute Vendor, Price and Date of Product. The result is given in Figure 6-18.

G.Product			
Npro	**Vendor**	**Price (Euro)**	**Date**
1	Michaux	109.58	1/01/2002
2	Gide	10	1/05/2002
3	null	99.16	1/06/2002

Figure 6-18: Conflict-free instances of the global entity type.

6.4 Reverse-Engineering Process and Model Translation

This is the process of extracting the legacy physical schema (LPS) and the legacy conceptual schema (LP_cS) of a legacy database (Figure 6-19). Recovering the physical and conceptual

schemas of an existing local database is the main goal of Database Reverse Engineering (DBRE). Our approach relies on the general DBRE methodology that has been developed in the DB-MAIN project ([Hainaut, 1996b], [Hainaut, 2002]).

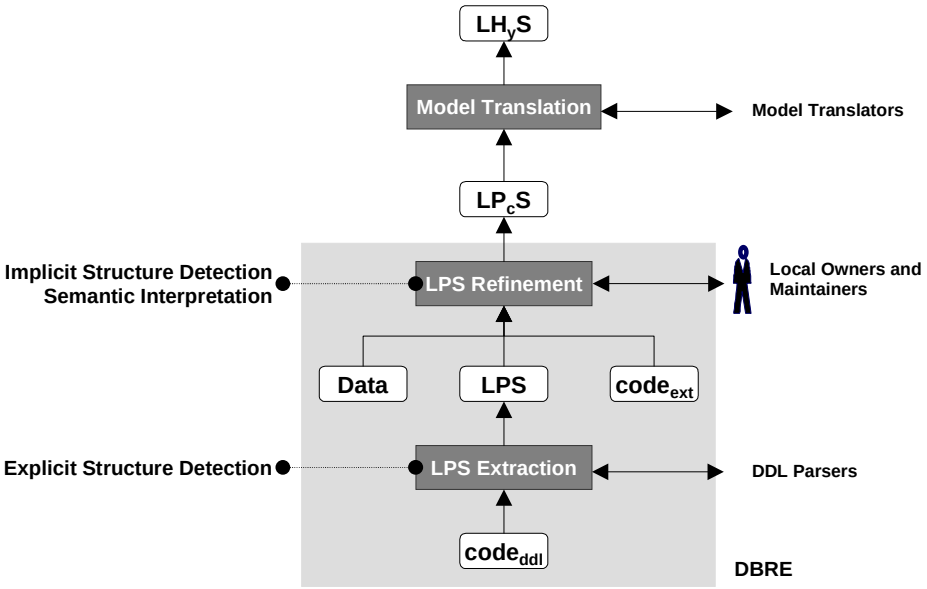


Figure 6-19: Reverse engineering of a legacy database. For simplicity, the mapping definitions are ignored.

As legacy databases can employ different data models in order to represent real-world entities, they need to be brought into a common formalism, so that they can be compared. Activities in this phase are performed for each legacy database only once when the source enters the federation, or when it decides to share and exchange information units with other components.

6.4.1 LPS Extraction

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, DDL texts, file sections, etc.). The process consists in analyzing the data structure declaration statements or the contents of these sources ($CODE_{ddl}$). 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.

6.4.2 LPS Refinement

The goal of this phase is to recover the complete physical schema, including all the implicit and explicit structures and constraints. As explained in Chapter 4, the main problem of the LPS refinement process is to discover and make explicit the structures and constraints that were either implicitly implemented (code_{ext}) or merely discarded during the development process (Δ). In this section, we analyze the problem and elicitation techniques of implicit constructs. We then present the most frequent implicit constructs and the transformation operators that make them explicit.

Implicit construct detection

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 evidence 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: namely, the *program analysis*, the *data analysis* and the *schema analysis*.

- *Program analysis.* This process consists in analyzing parts of the application programs (the procedural sections, for instance) in order to detect evidence of additional data structures and integrity constraints. The way data are used, transformed and managed in the programs brings essential information on the structural properties of these data. For instance, through the analysis of data validation procedures, the analyst can learn what the valid data values are, and therefore what integrity constraints are enforced. This kind of search is called usage pattern analysis. Being large and complex information sources, programs require specific analysis techniques and tools. Dataflow analysis, dependency analysis, programming *cliché* analysis and program slicing are some examples of program processing techniques that resort to the domain of program understanding.
- *Data analysis.* The data themselves can exhibit regular patterns, or uniqueness or inclusion properties that provide hints that can be used to confirm or disprove structural hypothesis. The analysis can find hints that suggest the presence of identifiers, foreign keys, field decomposition, optional fields, functional dependency, existence constraints or that restricts the value domain of a field for instance. 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.

Semantic interpretation

This process addresses the semantic interpretation of a legacy schema, from which one tries to extract the conceptual view. The objective is to identify and to extract all the relevant semantic concepts underlying the legacy 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 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 [Batini, 1992].

6.4.3 Model Translation

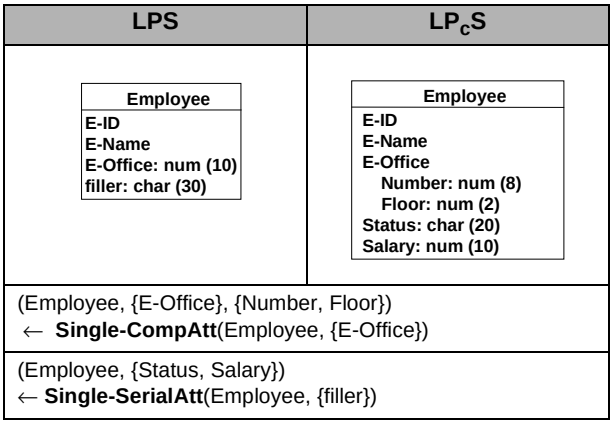
The model translation is a particular case of schema transformations (Chapter 3, Section 3.7). It consists in translating a schema expressed in a data model M_s into a schema expressed in another data model M_t where M_s and M_t are two different submodels (i.e., subsets) of the generic data model. As already mentioned in Chapter 4, this model transformation can be modeled as semantics-preserving schema transformations. This process can therefore be easily automated (Chapter 7).

6.4.4 Some Implicit Constraints and Constructs

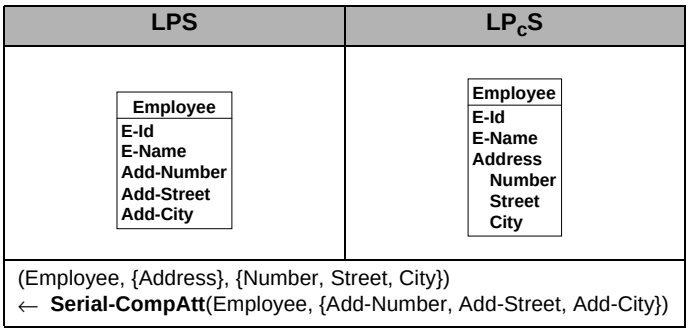
The variety of implicit constructs can be fairly large, even in small systems. This section presents the main implicit structures and constraints we found in reverse engineering processes ([Hainaut, 1996] and [Thiran, 2002d]. We then present the transformation operators that make them explicit and the constraint assertions associated with the implicit constraints.

Implicit compound attribute

A field, or a full record type, declared as atomic, has an implicit decomposition, or is the concatenation of contiguous independent fields (E-Office, 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.

LPS	LP_cS
Salesperson **Number** **Name** **Phone[0-1]: char (36)**	**Salesperson** **Number** **Name** **Phone[0-3]: char (12)**
(Salesperson, {Phone}) ← **Single-MultiAtt**(Salesperson, {Phone})	

LPS	LP_cS
Salesperson **Number** **Name** **Phone1** **Phone2[0-1]** **Phone3[0-1]**	**Salesperson** **Number** **Name** **Phone[1-3]: char (12)**
(Salesperson, {Phone}) ← **Serial-MultiAtt**(Salesperson, {Phone1, Phone2, Phone3})	

Implicit identifiers

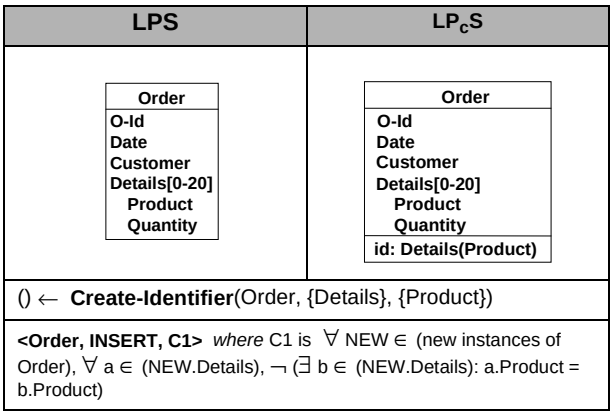
The identifier (or unique key) of a record type is not always declared. Such is the case for sequential files for example. The fact that the E-Id values are unique among the Employee records must be proved.

LPS	LP_cS
Employee **E-Id** **E-Name** **E-Address**	**Employee** **E-Id** **E-Name** **E-Address** **id: E-Id**
() ← **Create-Identifier**(Employee, {E-ID})	
<Employee, INSERT, C1> *where* C1 is ∀ NEW ∈ (new instances of Employee), ¬ (∃ OLD ∈ (old instances of Employee): NEW.E-Id = OLD.E-Id)	

Implicit identifiers of multivalued attributes

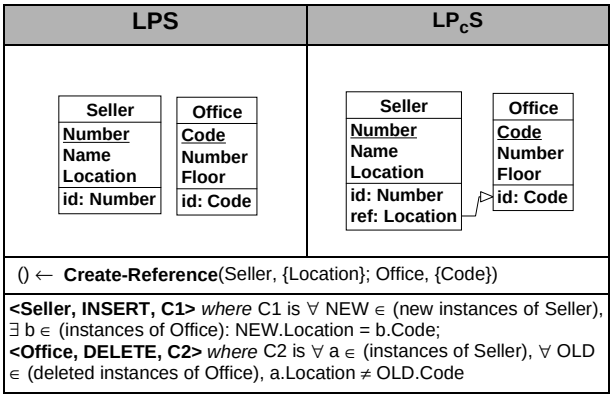
Structured record types often include complex multivalued compound fields. Quite often too, these values have an implicit identifier. In each Order record, there are no two Details values

with the same Product value.



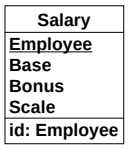
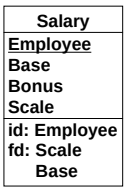
Implicit foreign keys

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.



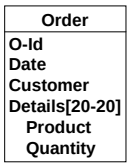
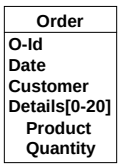
Implicit functional dependencies

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 Salary record type, the value of field Base depends on Scale, which is a non-key field. This record type is in 2nd normal form only.

LPS	LP_cS
	
() ← **Create-FunctionalDependency**(Salary, {Scale}; {Base})	
<Salary, INSERT, C1> *where* C1 is ∀ NEW ∈ (new instances of Salary), ¬ (∃ a ∈ (instances of Salary): (a.Scale = NEW.Scale) ⇒ (a.Base ≠ NEW.base))	

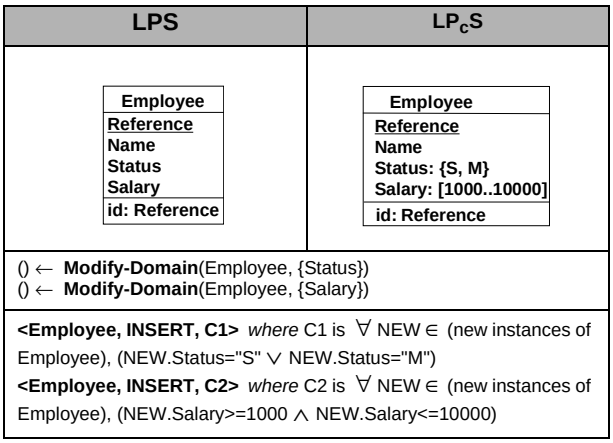
Finding exact minimum cardinality of fields and rel-types

Multivalued fields are generally declared 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 Details has been declared "occurs 20", and its cardinality has been interpreted [20-20]. Further analysis has shown that this cardinality actually is [0-20].

LPS	LP_cS
	
([20]) ← Modify-MaxCardinality(Order, {Details})	

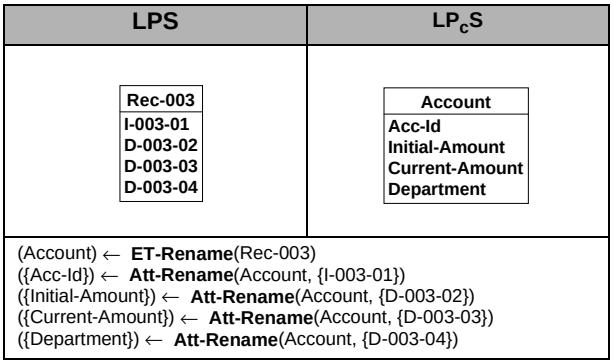
Implicit constraints on value domains

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 admittable values. In the example below, field Status has an enumerated domain, comprising two values "S" and "M", while values of field Salary must fall into the interval [1000...10000].



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.



6.4.5 Application to the Case Study

By analyzing the COBOL programs of DB-P1 and the SQL DDL scripts of DB-P2 and DB-S we can extract the Local Physical Schema (LPS) of each legacy database. Figure 6-20 shows the extracted schemas according to their data model.

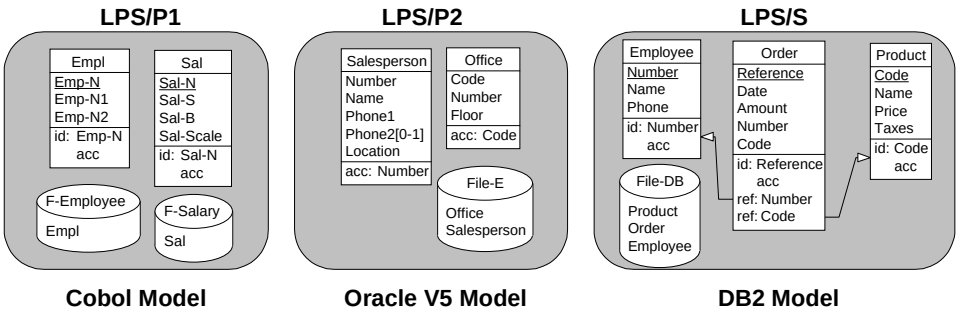


Figure 6-20: The local physical schemas. LPS/P1 is made of two files (F-Employee and F-Salary) and two record types (Empl and Sal). Emp-N and Sal-N are unique record keys. LPS/P2 comprises the tables Salesperson and Office. No primary key and no foreign key are represented in this schema since Oracle V5 model ignores these concepts. LPS/S comprises three tables Employee, Order and Product. The table Order has two foreign keys that reference the other two tables.

The LPS extraction is fairly straightforward. However, it recovers explicit constructs only, ignoring all the implicit structures and constraints that have been buried in, e.g., the procedural code of the programs. Hence the need for a refinement process that cleans the physical schema and enriches it with implicit constraints elicited by techniques described above.

By applying the schema refinement, we obtain the refined physical schemas of Figure 6-21, that makes 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. The schemas are refined through in-depth inspection of the data, and of the way in which the COBOL program and SQL DML use and manage the data in order to detect the record structures declared in the program sources. For instance, the Oracle V5 DB-P2 database includes a hidden foreign key that can be discovered by looking for, e.g., join-based queries. Moreover, names have been made more meaningful and physical constructs are discarded. We therefore obtain the three LP_cS of Figure 6-21. Finally, all the LP_cS are translated into the binary ER model used as the canonical data model (Figure 6-22).

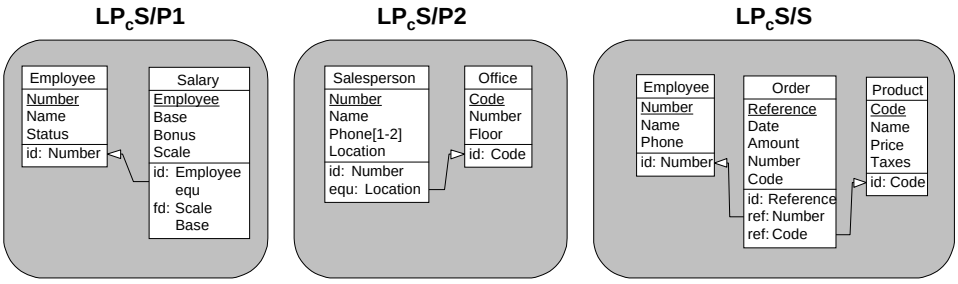


Figure 6-21: The refined physical schemas. In LP$_c$S/P1, we observe the renaming and the elicitation of an implicit foreign key and an implicit functional dependency. In LP$_c$S/P2, two hidden identifiers, a hidden foreign key and an implicit multivalued attribute have been discovered. The purely physical objects have been removed and names have been reworked.

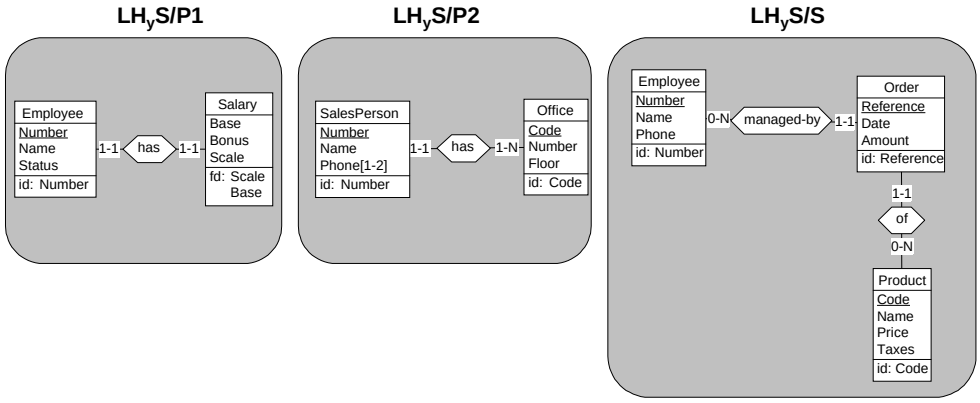


Figure 6-22: The syntactically homogenized legacy schemas (LH$_y$S). LP$_c$S have been modeled using the binary ER model. The four foreign keys of the three LP$_c$S have been transformed in relationship types by applying the semantics-preserving transformation FK-RT.

6.5 NGS Definition and Homogenization

The purpose of this process is to define (1) the new global schema (NGS) that captures the complete requirements of an organization for the future; and (2) the homogenized legacy schemas (LH$_g$S) that contain the relevant legacy information (Figure 6-23).

An important characteristic of the approach is the role of the legacy databases in the require-

ment analysis process. Indeed, since the existing system meets, at least partially, the current needs of the organization, it is an invaluable source of information about the requirements of the future system. On the other hand, the homogenization phase can easily identify the constructs of a LH_yS that can be reused in the future system.

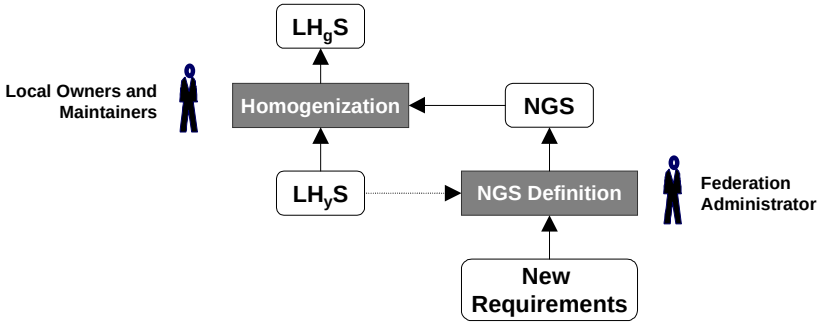


Figure 6-23: The analysis process is made up of two subprocesses: the new global schema (NGS) definition and the homogenized legacy schema (LHgS) definition.

6.5.1 NGS Definition

Collecting and formalizing the requirements of the organization into a conceptual schema is a standard, though still complex, process that has been widely described in the literature. As an implementation of a part of the requirements that are being elaborated, the semantic description of the legacy databases, i.e., their conceptual schemas (LH_yS) can contribute to the NGS definition. The importance of this source of information has been acknowledged and developed in [Batini, 1992] and [Blaha, 1998] for instance.

6.5.2 Homogenization

After recovering the LH_yS, knowledge required to interrelate them is acquired in a stepwise fashion during the homogenization phase. Acquired knowledge is used to obtain a modified local database schema (LH_gS) that is more tightly coupled with NGS than that of the previous version of the LH_yS.

Homogenization removes derivations of a source from NGS; these discrepancies arise when a LH_gS and NGS model the same application domain differently [Yan, 1997].

Homogenization involves only one database source and compares it with NGS. Discrepancies resolved during the homogenization phase are due to differences between the definition of a pair of schemas (LH_yS and NGS). As a result, it refers to the intensional dimension of the legacy database only.

NGS definition strategies

NGS reflects design efforts to accommodate the conflicting legacy and new requirements. However, different organization goals are possible and can lead to different NGS definition.

- *NGS dominates*[2] *the local schemas*. NGS is only defined by the new requirements. An authority forces all the local sources to make their legacy databases comply with the new business requirements that it has defined itself. Each local source is responsible for expressing the sharable portion of its local schema in terms of NGS. The mismatches must therefore be resolved at the local level.
- *NGS and the local schemas agree* with their commonly understood set of concepts. Both NGS and the local schemas can be modified/restructured in order to reduce their syntactic and semantic distance. In this approach, each local source plays the role of an information source for the NGS definition.

Discrepancy resolution

We state that most discrepancies can be solved through four main techniques that are used to rework the LH_yS according to NGS, namely, renaming, generalizing, transforming and discarding.

- *Renaming.* Constructs that denote the same application domain concepts are given the same name.
- *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.
- *Transforming.* An application domain concept can be represented by constructs of different nature in a LH_yS and NGS. A supplier can be represented by an entity type in NGS and by an attribute in a local source The latter construct will be transformed into an entity type to give both representations the same nature.
- *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.

Note that some of these techniques can not be applied when NGS dominates the local schema since they require an agreement between NGS and LH_yS. This is the case for the generalization of cardinality conflicts.

6.5.3 Some Typical Discrepancies

As for implicit constructs, the number and variety of discrepancies can be fairly large. We

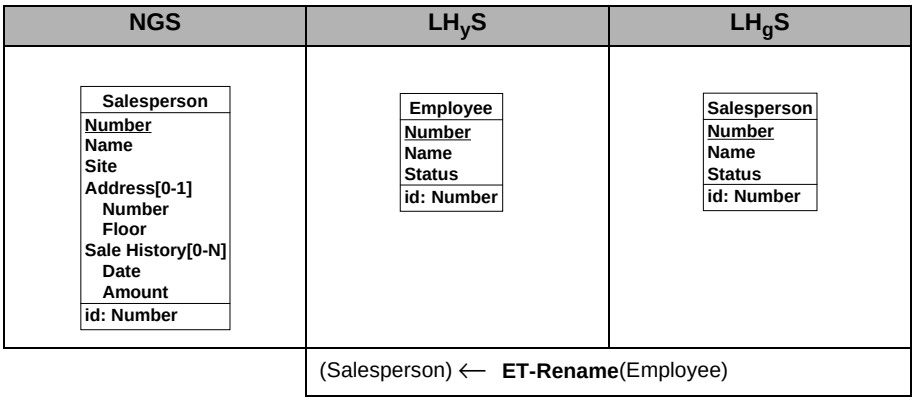
2. The term *dominates* is defined as follows: any instance of the dominant schema is an instance of the dominated schema.

classify those discrepancies which are to the fore for our approach. Starting which syntactic conflicts for an overall view, we relate them to semantic ones and discuss basic techniques for their resolution. Their resolution is demonstrated by examples in the rest of this section. In each step, derived mappings are specified as schema transformations.

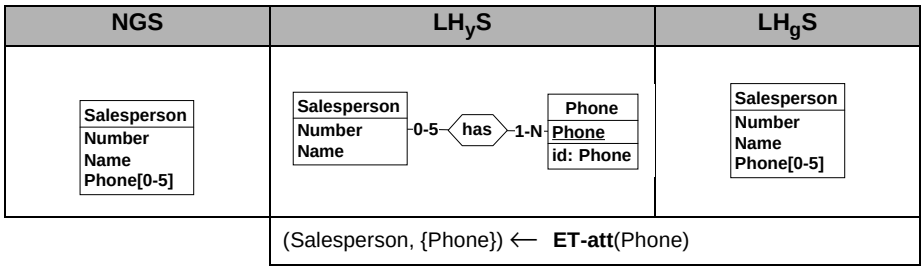
Syntactic discrepancy

Besides the usual conflicts related to synonyms and homonyms, a syntactic discrepancy occurs when the same concept is presented by different constructs in local schemas. For instance, a same construct can be represented by an entity, by an attribute value and by a relationship. These syntactic mismatches are resolved by transforming LH_yS with respect to NGS. While NGS is not modified, LH_yS is transformed by means of semantics-preserving transformations.

Naming mismatches. Naming 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. In our approach, we apply a renaming transformation to the construct of LH_yS with respect to the same name that its synonym in NGS.



Structural conflicts. Relationship types in NGS can be modeled by different schema constructs in the local databases. For example, at the conceptual level, a real world association can be represented as a relationship type in one local database but as an entity type in NGS.



Semantic discrepancy

A semantic discrepancy appears when a contradiction appears between two representations A and B of the same application domain concept or between two integrity constraints. Unlike structural discrepancy, a semantic discrepancy can not be resolved by means of semantics-preserving schema transformation.

Semantics conflict detection requires knowledge about the problem domain, the local schemas and the extensional dimension of the legacy databases. 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.

There are three strategies that can be used for resolving a semantic discrepancy (Figure 6-24):

- *Generalizing.* If two constructs denote the same application domain concept, and if one of them is more constrained, the constraint is relaxed.
- *Construct transforming.* Transformations of entities/attributes/relationships among each another are another way of handling semantic conflicts. These transformations can keep the semantics of each schema and can also create other discrepancies.
- *Reconciliation function.* An appropriate user-defined function can be used to compute the value of any constructs conflicting with other corresponding constructs. In this case, the user-defined function is most likely to be a table lookup. According to this strategy, NGS is left unchanged.

Strategy	Transformed Schema	Transformation Type	Structural Equivalence
Generalizing	NGS and LH_yS	schema transformation	yes
Transforming	NGS and LH_yS	schema transformation	no
Reconciliation function	LH_yS	user-defined function	yes

Figure 6-24: Strategies of semantic conflict resolution and their characteristics.

We illustrate the strategies of the semantic conflict resolution by developing some standard semantic discrepancies such as identifier conflicts, cardinality conflicts, missing attributes

and generalization conflicts.

Identifier conflicts. In this case, the entity type descriptions are incompatible because they use identifiers that are semantically different. There are several ways of resolving this kind of conflicts. In Figure 6-25, we present three different solutions for resolving a conflict between the identifiers of two structurally equivalent entity types.

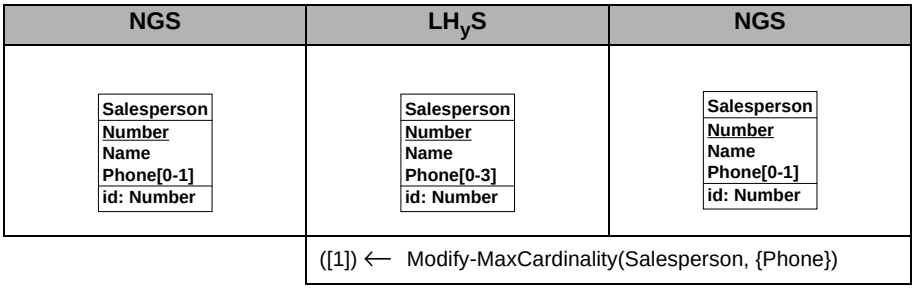
NGS	LH_yS
ET A1 A2 A3 id: A1	ET A1 A2 A3 id: A2

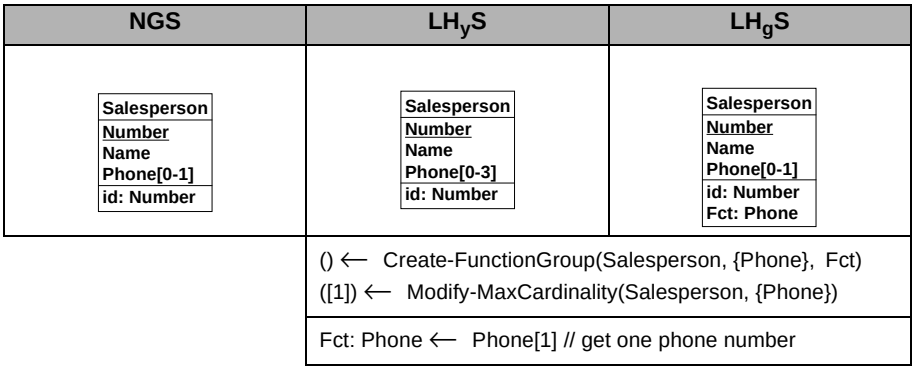
	NGS	LH_gS
Generalization: the identifier group is made up of the two identifiers. NGS and LH_gS are modified into a common structure. The transformations are not SR-transformations.	ET A1 A2 A3 id: A1 A2	ET A1 A2 A3 id: A1 A2
Att-ET/Value: A non-identifier attribute is transformed into an identifier of an other entity type. NGS and LH_gS are modified by means of schema transformations. The resulting structures are semantically different.	ET A1 A3 id: A1 1-1 R 1-N ET' A2 id: A2	ET' A2 A3 id: A2 1-1 R 1-N ET A1 id: A1
User-defined function: a table maps the conflicting attributes to appropriate values. While NGS is not touched, LH_gS is modified by means of reconciliation function.	ET A1 A2 A3 id: A1	ET A1 A2 A3 id: A1 Fct: A1 A2 A3

Figure 6-25: Identifier conflict resolution.

Cardinality conflicts. Cardinality conflicts arise when the local databases have different cardinalities for the same relationship type or the same attribute in the integrated schema. For example, an attribute can be represented as a a single-valued attribute in NGS whereas it is defined as a multivalued attribute in a legacy database (LH_yS). The cardinality conflict can be resolved in two ways:

- by using the generalizing strategy if NGS can be modified;
- by defining a reconciliation function that is able to select one of the instances of the multivalued attribute. This strategy leaves NGS unchanged.





Missing attributes. Semantically similar entities can have a different number of attributes. Corresponding entity types can be described by different attributes in the local schemas and NGS. This is due to different requirements of the legacy and new applications. In one legacy system, local applications can need a certain attribute of the entity type whereas in the new system, no application requires this attribute. This refers to the *missing attribute* in similar entity types in different schemas [Elmagarmid, 1999]. Intuitively, the solution is to compare each LH_yS against NGS in order to keep the common attributes only. Actually, there are two possible cases:

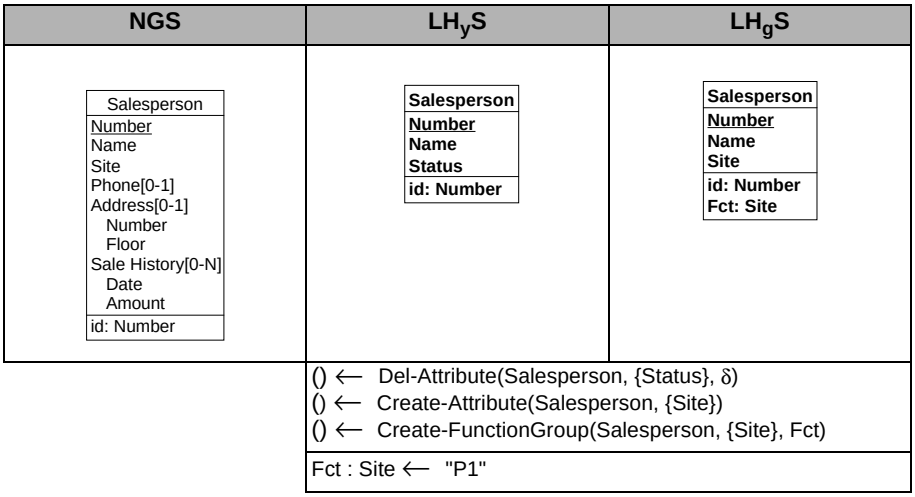
- An attribute of an entity type of LH_yS is missing in NGS, then it is simply discarded.

- An attribute of an entity type of NGS is missing in LH_yS.

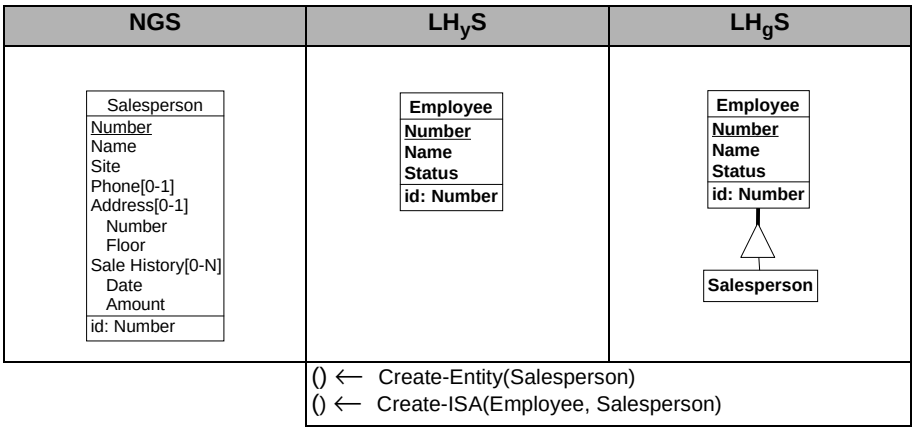
A special case of the conflict above satisfies the following conditions:

- The missing attribute is compatible with an entity.
- There exists an inference mechanism to deduce the value of the attribute.

Consider the two schemas NGS and LH_yS below. LH_yS does not have an attribute Site but that can be implicitly deduced to be P1 (the name of the local site). In the above example, Salesperson of LH_yS can be thought to have an attribute Site whose default value is "P1". The mapping is done in such a way that the value of the attribute can be inferred from the relationship specified between the two terms.



Generalization Conflicts. These conflicts arise when two entities are represented at different levels of generalization in NGS and LH_yS. Consider that NGS only considers employees that work as a salesperson whereas the local database records all the employees that work in its site whatever their status. Thus, we have the same concept being defined at a more general level in LH_yS. In our approach, the term associated with the more general entity type must *subsume* the term associated with the more specific entity type by means of adding the selection construct into a query posed on that entity type. In case the constructs have different names then a renaming transformation must be performed.



Domain mismatches

In this section, we discuss incompatibilities that appear when domains of two different types are used to defined domains of semantically similar attributes [Kashyap, 1997].

Further examples for description conflicts are that corresponding attributes can have different data types or ranges in different legacy systems. Even if they have the same data type, different units of measurement or different scaling can be used within the legacy systems.

Data representation conflicts. Two attributes that are semantically similar might have different data types or representations. We define used-defined conversion functions which convert data from the legacy representations to the new ones.

Data scaling conflicts. Two attribute that are semantically similar can be represented using different units and measures. There is a one-to-one mapping between the values of the domains of the two attributes. For instance, the salary attribute can have values in Belgian Francs or Euros. We apply a schema transformation that specifies and defines transformation functions which convert data from the different scales to the common scale subscribed to by the NGS.

Data precision conflicts. Two attributes that are semantically similar can be represented using different precisions. This case is different from the previous case because there are not one-to-one mapping between the value domains. There can be a many-to-one mapping from the domain of the precise attribute to the domain of the coarser attribute. Let the attribute mark have an integer value from 1 to 100. Let attribute grade have the values {A, B, C, D, E and F}. We must define conversion table to convert data to the precision of measurement used by the NGS. In this case, the functions can be based on a table lookup (Figure 6-26).

Mark	Grade
81-100	A
70-79	B
60-69	C
50-59	D
40-49	E
1-39	F

Figure 6-26: Example of a mapping table.

Default value conflicts. This type of conflict depends on the definition of the domain of the concerned attributes. The default value of an attribute is that value which is defined to have in the absence of more information about the real world. For instance, the default value for Age of an adult can be defined as 18 years in one database and as 21 years in another.

6.5.4 Application to the Case Study

Global Conceptual Schema Definition. The NGS is defined from the new requirements of the information system and the information held by the personnel databases and sales database. In this way, the NGS integrates legacy information and new information about salespersons, customers and their orders (Figure 6-27).

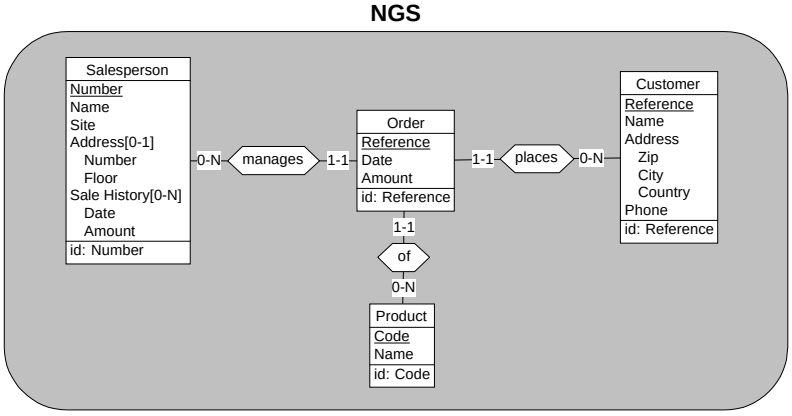


Figure 6-27: The new global (conceptual) schema that formalizes the requirements to be met by the future system.

Homogenization. By removing, from each LH_yS, the structures and constraints that are not in NGS and by homogenizing them according to NGS, we derive the three LH_gS (Figure 6-

28).

Figure 6-28 depicts some discrepancies between LH_yS of the two sources and NGS: (1) *domain discrepancy:* in LH_yS, salesperson numbers are recorded with dash separation while the same data are to be without any dash in NGS; (2) *syntactic discrepancy:* Address and Office are synonyms; (3) *semantic discrepancy*: Office is modeled as an entity type in LH_yS of the site P2 while it is modeled as a compound attribute in NGS; NGS only considers employees that work as seller whereas the local database records all the employees that work in the site P1 whatever their status (the value domain of Status of Employee is {Salesperson, Secretary, Manufacturer, Manager}.

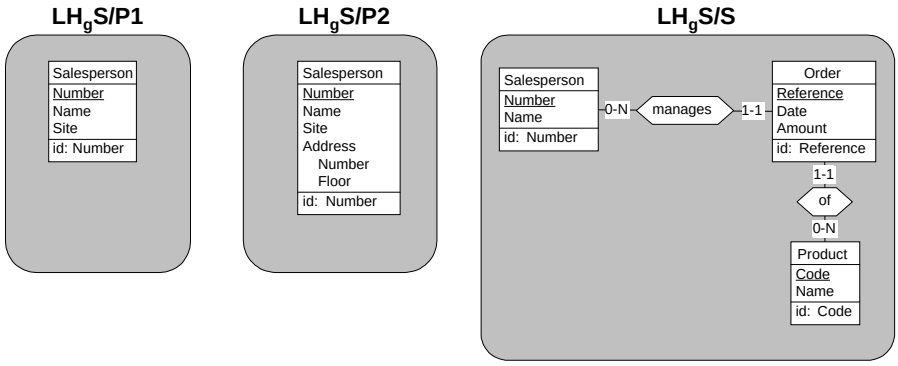


Figure 6-28: The legacy homogenized schemas. In LHgS/P1, the entity type Employee has been renamed Salesperson and only the attributes Number and Name are kept. In LHgS/P2, the entity types Salesperson and Office have been transformed into one complex entity type Salesperson. The entity types Salesperson of these both schemas include a calculated attribute Site that returns the site of the Salesperson. LHgS/S contains all the structures of LHyS/S except for the entity type Product.

6.6 Legacy-Legacy Integration

The objective of this process is to build the hierarchy of integrated schemas (Figure 6-28) that is to form the legacy global schema (FGS).

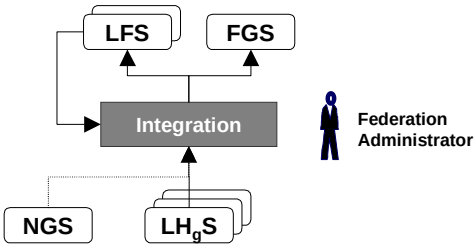


Figure 6-29: Legacy-legacy integration.

6.6.1 Principles

Based on the homogenized view of legacy databases, legacy-legacy integration is the process of merging them. Identifying similar constructs and merging them is quite easy thanks to the results of the previous steps. Indeed, the reverse engineering step has given the analysts a strong knowledge of the semantics of each source construct. In addition, the homogenization process has already determined the horizontal correspondences and produced a a view of the legacy databases which has been homogenized in regard to NGS. 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.

The results of this process are (1) the legacy global schema (FGS), (2) the set of intermediate legacy federated schemas (LFS), and (3) the mapping definition between the hierarchy schemas. Several problems have to be solved. One of them is determining the optimal structure of the federated schema hierarchy. Another is the schema integration and the resolution of data inconsistency.

6.6.2 Schema hierarchy

Defining the intermediate nodes in the hierarchy is an important issue that has yet to be worked out. More specifically, the question is:

> *When have two (or more) databases to be integrated into a common schema, and managed by the same mediator?*

Though we have no stable answer to this question yet, we can identify four major dimensions and criteria according to which legacy databases can be grouped and integrated:

- *Similarity of organizational objectives.* Both databases focus on the same part of the application domain, so that merging them mimics the merging of these similar organization units.

- *Complementarity of organizational objectives.* The databases control two different parts of the application domains that share some objects, or such that objects from one of them are related with objects from the other one. Merging these databases parallels the merging of two complementary organizational units.
- *Locality.* The databases are located on the same site or on the same server. Merging them produces a virtual database that is identified by its location.
- *Technology.* Two databases are under the control of a unique DBMS. Merging them can be done by a technically homogeneous mediator.

6.6.3 Integration Process

Steps generally addressed in theoretical approaches to schema integration are of a lesser importance here since they have been performed in the reverse engineering and homogenization processes. This is the case for conflict and horizontal correspondence identification and resolution.

Horizontal correspondence detection

Since the structural and semantic distance between each local source and NGS has been reduced during the previous step, the similar parts of the schemas have almost identical representations. We can therefore use simple denotation assumptions like the following ones:

- 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.

Data inconsistency

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. Such a conflict covers incompatibilities that arise due to the values of the data present in different databases.

These conflicts are different from discrepancies resolved during the homogenization phase in that the latter are due to the differences in schema definitions. Here, we refer to the data values already existing in the database. Thus, the conflicts depend on the database state. Since we are dealing with autonomous databases, it is not necessary that the data values for the same entities in two different databases be consistent with each other. Sources for instance conflicts include typing errors, variety of information providers, different versioning, deferred updates ([Parent, 2000] and [Kashyap, 1997]). These conflicts are normally found during query processing. The system can just report the conflicts to the user, or can apply some heuristic (reconciliation function) to determine the appropriate value.

We identify three kinds of instance conflicts; namely, identifier equivalence conflicts, at-

tribute value conflicts, relationship conflicts:

- *Identifier equivalence conflicts* arise when instances from different entities refer to the same real-world object but contain different identifier values.
- *Attribute value conflicts* occur when instances, which correspond to the same real-world object and share an identifier, differ in other attributes. One reason for this problem could be a situation, where two entities from different sources overlap semantically and one of the entity contains older or outdated data.
- *Relationship conflicts* occur when a same relationship type represents associations between different entities.

Example

Considering the entities Salesperson recorded in the databases P2 and S (Figure 6-30). For simplicity, we only consider the two common attributes among these entity types, namely: number and name.

Salesperson/P2		
Number	**Name**	**Phone**
AA1	Michaux	724981
AA2	Valéry	null
AA3	Gide	724967

Salesperson/S		
Number	**Name**	**Phone**
AA1	Michaux	724981
AA2	Valéry	724981
AA3	Gide	724981

Figure 6-30: Example of attribute value conflicts.

Attributes conflicts are resolved by specifying a user-defined reconciliation function for the merging operation. The reconciliation function implements a rule based on the fact that P1 refers to the most recent phone numbers of salespersons. Therefore, if the phone number of a salesperson in P1 is not null then, it should be used otherwise the phone number value of the same salesperson from S appears in the integrated results.

6.6.4 Application to the Case Study

From the horizontal correspondences defined in the previous step, the integration process is fairly easy. We define two federated schemas as follows (Figure 6-31).

- The databases DB-P1 and DB-P2 describe the same application domain (HRM). Therefore, they meet the similarity criterion, so that we suggest to define a virtual personnel database encompassing both department needs, with schema LFS/P.
- This database has some relationship with the sales database through the concept of salesperson. We propose to merge the virtual personnel database with the sales database. The common schema is also the global description of the federation. It is named FGS/PS.

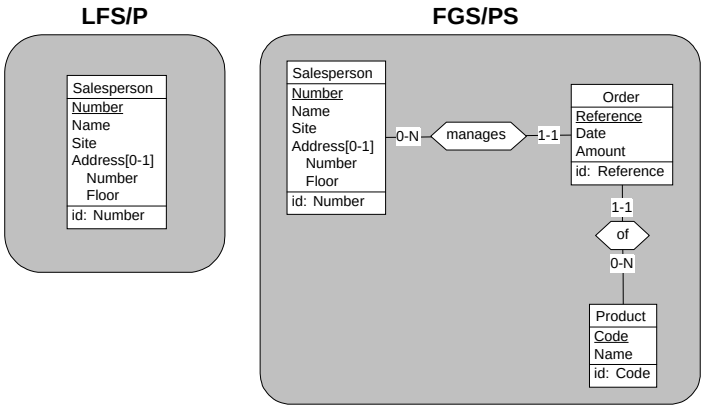


Figure 6-31: The conceptual federated schemas. LFS/P integrates the LHgS of the personnel databases whereas FGS/PS integrates LFS/P and the LHgS of the sales department. For readability, the user-defined function groups are not represented.

6.7 Global-Legacy Comparison

6.7.1 Principles

The global-legacy comparison consists in comparing NGS that holds the information required by the whole system against the FGS that integrates the existing information (Figure 6-32). Intuitively, the conceptual schema of the new database is obtained by *substracting* the concepts of FGS from NGS (Chapter 4, Section 4.2.7). Since FGS integrates LH_gS defined in the same process than the NGS, the FGS is likely to be a subset of NGS, which simplifies the derivation in many cases. The resulting schema is called the new schema (NH_gS). It holds the information that is in NGS but not in FGS.

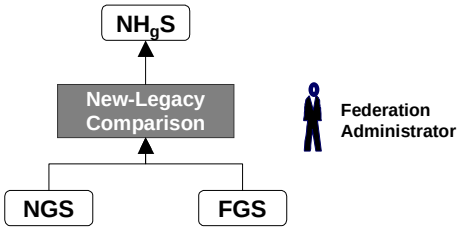


Figure 6-32: Global-legacy comparison. This process produces the starting point for the development of the new database.

Building an operational database from this conceptual schema is a standard problem [Batini, 1992]. The usual architectural issues and problems of distributed databases generally will appear and have to be solved. However, some specific issues must be addressed, related with the fact that this new database has to operate in collaboration with the federated database. For instance, some of the important entity types that appear in the federated database may also appear in the new database, either as duplicates (e.g., to represent additional attributes of an entity type, or to improve performances), or as extension (e.g., the personnel of a third department has to be incorporated) of the former ones.

6.7.2 New Database Definition

Before we proceed, let us clarify the *substracting* concept introduced above. It consists on analyzing NGS and marking the constructs that are non-overlapping with FGS. The unmarked constructs are discarded. Non-overlapping constructs refer only to constructs that do not participate in an identifier group. Therefore, substracting is considered only for those constructs that do not participate in a identifier group. The identifiers are kept as *references*, i.e., links between new and legacy databases. The rules for substracting NGS against FGS can be expressed as follows:

- Mark the constructs that are in NGS but not in FGS.
- If the marked construct is a RT, then mark all the identifiers of the ET which participate in the RT.
- If the marked construct is a sub-type, then mark the identifier of its supertype.
- If the marked construct is an attribute, then mark the ET it belongs to and its identifier.

The application of these rules gives a minimal NH_gS that can be enriched with other constructs that appear in the federated database, either as duplicates or as extension.

Note that references cause problems such as loading (Section 6.7.3) and refreshing (Section 6.7.4) the new database. This concerns the synchronization of local updates without touching in the autonomy of legacy databases.

6.7.3 Data Extraction and Loading

Data extraction and loading aim at extracting needed data from the database federation and populating the target new database. The legacy data are extracted and loaded initially into the new database when the new database is created. Typically, data extraction loading is taken in charge by a software, called an *Extract-Transform-Load* (ETL) processor ([Umar, 1997], [Delcroix, 2000]) that can use the federated database (i.e., wrappers and mediators) for accessing the legacy databases through FGS (Figure 6-33).

An ETL processor has three main functions. Firstly, it performs the extraction of the needed data from a legacy database. Then, it converts these data in such a way that their structures match the target format. Finally, it writes these data in the target database.

Since the mappings between the source schema (FGS) and the target schema (NH_gS) can be

formally defined by means of schema transformations [Delcroix, 2000], an ETL processor can rely on the schema transformation approach to automate the data migration. In such a context, data migration involves two main tasks. Firstly, the mapping between the source and the target schemas must be defined as sequences of schema transformations (FGS-to-NH_gS). Second, these mappings must be implemented in the ETL processor for translating the legacy data according to the format defined in NH_gS. An ETL processor relies on the structural mappings to write the extraction and insertion request, and on the instance mappings for the data conversion. We refer to [Delcroix, 2000] for a development of an ETL processor based on the principles described above and that use our wrapper technology.

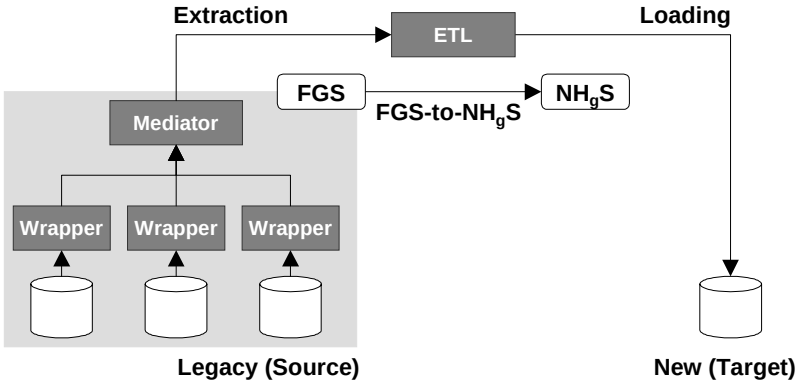


Figure 6-33: Data migration architecture: ETL based on the schema transformation sequence between FGS and NHgS.

6.7.4 Data Refreshing

Data refreshing still presents serious challenges [Özsu, 1999]. This concerns the freshness of data of the duplicates. Data can be updated at multiple sites independently even if this leads to inconsistent replicates.

Replication techniques have been used successfully for refreshing a data warehouse periodically [Helal, 1997]. As far as legacy systems are concerned, a challenging problem is the change detection, which detects and propagates the changes in the legacy sources to the new sources. Legacy databases can be classified according to their ability for change detection [Widom, 1995]. Cooperative sources provide trigger capabilities which ease the programming of automatic notifications of changes. Logged sources maintain a log from which changes can be extracted. Queryable sources can be queried by the new system. Finally, snapshot sources can only be copied off-line. The WHIPS data project [Hammer, 1995] addresses the problem of automatic change detection and incremental data integration.

6.7.5 Application to the Case Study

By comparing the NGS/PSC against FGS/PS, we define NH_gS, the schema that holds all the new constructs and constructs (Number and OrderReference that links the customers and their orders) common to the FGS/PS (Fig. 6-34). Here, the connection with the federated database is made possible through a common identifier and a reference to an identifier .

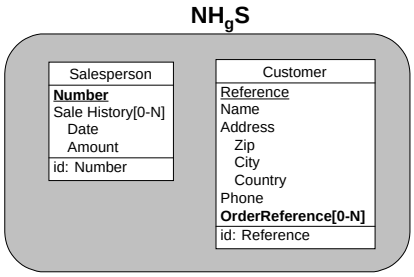


Figure 6-34: The local conceptual schema of the new database.

Part IV

CASE Support

Chapter 7

CASE Tool Technology

In which we analyse the requirements that CASE tools should meet for the development of the schema-oriented framework of data mediation, and present a general architecture for such CASE environments. We also present an operational CASE tool which is intended to address some of these requirements.

7.1 Introduction

Like any complex process, developing the schema-oriented framework of data mediation cannot be successful without the support of adequate tools called CASE tools [Conrad, 1999]. Nevertheless, completely automating this process is unrealistic for real world systems ([Sheth, 1991], [Hainaut, 1999], [Sattler, 2003]). It is however possible to reduce the amount of human interaction [Elmagarmid, 1999].

Several CASE tools are now available for building database federation. They address several aspects of the development of database federation development. These aspects include integration methodology and software production:

- *Integration methodology.* [Hayne, 1992], [Gotthard, 1992], [Ramesh, 1995], [Castano, 1999], [Terracina, 2000] propose tools for automated interschema relationship identification. Many of these tools address the resolution of intensional conflicts and propose semi-automatic techniques for discovering synonyms, homonyms and other schema relationship. For instance, [van den Heuvel, 2002] presents techniques that detect and resolve intensional conflicts automatically by reasoning about semantics in a knowledge base or ontology. More recently, [Sattler, 2003] considers both intensional and extensional conflicts and proposes basic facilities for their detection and reconciliation.

- *Software production.* [Papakonstantinou, 1995] proposes an implementation toolkit that facilitates the rapid development of wrappers and mediators. [Subrahmanian, 1995] and [Yan, 1997] provide 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 database federation development. For instance, many of these tools are based on the quality and completeness of the database structures to be integrated that cannot be relied on in many practical integration. In such tools, it appears that the 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 legacy ones.

Moreover, these proposals are most often dedicated to one data model (relational model for [Sattler, 2003] or [Yan, 1997]; OEM model for [Papakonstantinou, 1995]) and 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 database federation unanswered.

To give an answer to this question, we discuss some important requirements that should be satisfied by future CASE tools (Section 7.2). We then present an operational CASE tool (DB-MAIN) which is intended to address these requirements. In Section 7.3, we describe some of the original principles and components: its architecture, its repository, its user interface, its development language, its transformation toolkit and its history manager. The last sections describe in further detail its main assistants dedicated to specific aspects of the forward-reverse methodology (Section 7.4) and wrapper development (Section 7.5).

7.2 Requirements

This section states some of the most important requirements an ideal CASE tool environment for the development of database federation should meet. These requirements are induced by the analysis of the specific characteristics of the schema-oriented framework presented in Chapter 4. They also derive from the study of other CASE tool environments like VIbE [Sattler, 2003], the Aurora toolkit [Yan, 1997], the Tsimmis wrapper implementation toolkit [Papakonstantinou, 1995] and the Bales tools [van den Heuvel, 2002].

Data-centered activity

Developing a schema-oriented framework 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 (specifications management, evaluation, graphical viewing, reporting and code generation, etc.).

Schema transformation

Moreover, schema transformation is at the core of methodologies that manipulate schemas (Chapter 3). Therefore, automating these transformations increases the power and the reliability of the tool. The CASE tool must provide a rich set of transformation techniques.

Generic and flexible data model

The generic data model is designed to express all the semantics of the legacy data models (Chapter 2). Moreover, at any time, the current specifications can include constructs from different abstraction levels. For instance, a schema in process can include record types (physical objects) as well as entity types (conceptual objects). The specification model must be *wide-spectrum* and accommodate schema specifications at various levels of abstraction, and according to various common legacy paradigms.

Moreover, further information (e.g., transaction management or security) is necessary to build efficient wrappers/mediators. The CASE tool should maintain any type of information that can be used for specific need.

Source multiplicity

Extracting implicit structures and constraints; 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.

Schema history

The trace of the schema transformation sequence between the schemas are precisely and formally recorded in a history (Chapter 3, Section 3.6). The tool must therefore include a history recorder. It must also include various functions for history management: viewing at different aggregation levels, restructuring, reversing, analyzing, updating, annotating.

Flexibility

The methods of the schema-oriented framework are basically exploratory and often unstructured activities. Some important aspects of higher level specifications cannot be deterministically inferred. The tool must allow the user 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.

Extensibility

Each database federation is a new problem of its own, requiring specific reasoning and techniques. Integrating schemas appears to be a learning activity. The predefined functions should

be easy to customize and to program, and, specific functions should be easy to develop.

Interoperability

There are no available tools that can satisfy all corporate needs in development of database federations. In addition, current CASE tools already provide elaborated techniques that deal with some specific aspects of the design process (Section 7.1). The CASE tool must communicate easily with other development tools, exchanging specifications through common formats (such as XMI, or a common repository).

Genericity

Many integration reasonings and techniques are common to several processes and strategies. The basic techniques offered by the tool must be DMS-independent and therefore highly generic. Several integration strategies can be applied, depending on the complexity and the heterogeneity of the source databases and on skill of the analyst. The tools must include a collection of basic techniques for the integration instead of a unique, automated, schema integrator.

7.3 DB-MAIN

The DB-MAIN database engineering environment is a result of a research and development project initiated in 1993 by the database application engineering laboratory of the Computer Science Department of the University of Namur. This tool is dedicated to database applications engineering, and its scope encompasses, but is much broader than, database federation development alone. In particular, its ultimate objective is to assist developers in database design, database reverse engineering, maintenance, migration, integration and evolution. Further detail on the whole approach can be found in [Hick, 2002].

As a data-centered and wide-scope CASE tool, DB-MAIN is a repository-based environment that offers powerful toolkits and assistants to help developers and analysts carry out complex and tedious tasks in a reliable way (Figure 7-1 and Figure 7-2). These toolkits and assistants include transformation toolboxes, reverse engineering processors, schema analysis tools and integration assistants. In particular, DB-MAIN offers a rich set of transformational operators (including semantics-preserving ones) that allow developers and analysts to define the layer mappings in a systematic and formalized way.

Another interesting feature of DB-MAIN is its methodological control engine [Roland, 2003] and its meta-CASE layer, which allow method engineers to customize the tool and to add new concepts, functions, models and even new methods. In particular, DB-MAIN offers a complete development language, *Voyager 2*, through which wrapper generators have been developed and seamlessly integrated in the tool.

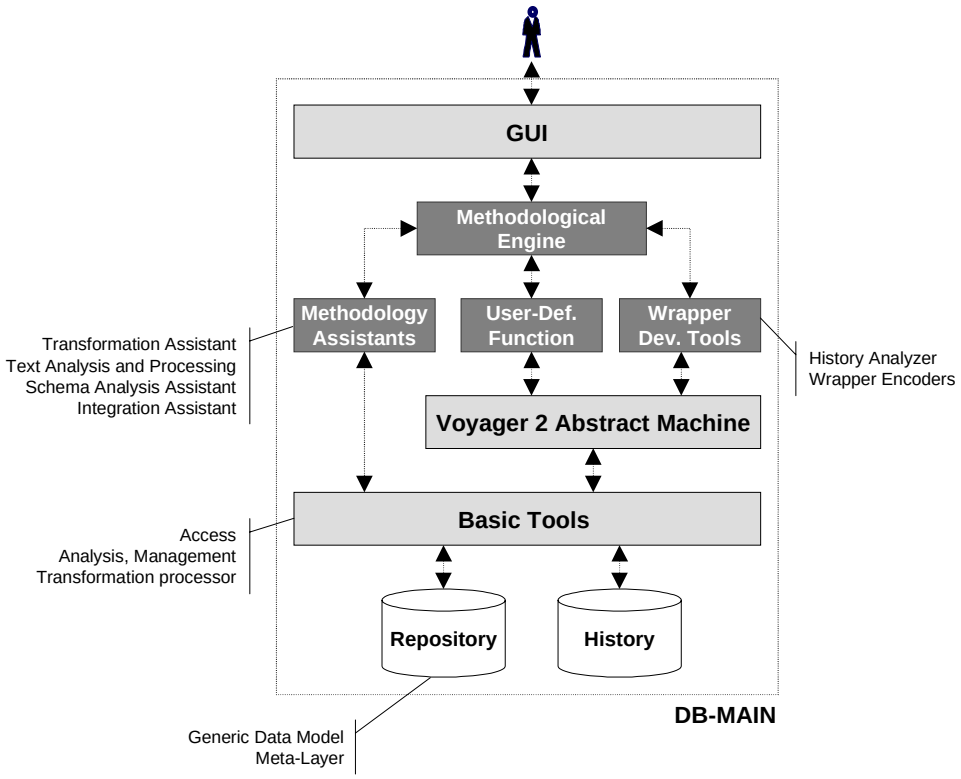


Figure 7-1: General Architecture of DB-MAIN CASE tool.

As we will see through this chapter, DB-MAIN presents some desirable properties such as flexibility, extensibility, openness and genericity:

- *Flexibility:* instead of being constrained by rigid methodological frameworks, the analysts is provided with a collection of neutral toolsets that can be used to process any schema whatever its level of abstraction and it degree of completion. In particular, backtracking and multi-hypothesis exploration are easily performed. However, by customizing the methodological engine, the analyst can build a specialized DBRE tool that is enforce strict methodologies, such as that which has been
- *Extensibility:* through the *Voyager 2* language, the analyst can quickly develop specific functions; in addition, the assistants allow the analyst to develop customized scripts.
- *Interoperability:* DB-MAIN supports exchanges with other CASE tools. XML-based tools are being developed (1) to generate specifications in XML, (2) to load into the repository the XML specifications produced by these tools.
- *Genericity:* both the repository schema and the functions of the tool are independent of the DMS and of the data model used in the databases to be analyzed. They can be used to

model and to process specifications initially expressed in various technologies. DB-MAIN includes several ways to specialize the generic features in order to make them compliant with a specific context.

Methodology	DB-MAIN Function	Section
DBRE	Transformation toolkit and assistant	7.3.4 and 7.4.1
	Text analysis and Processing	7.4.3
	Foreign key assistant	7.4.4
Model translation	Transformation and model-driven assistant	7.3.4 and 7.4.1
	Schema analysis assistant	7.4.2
Homogenization	Transformation toolkit and assistant	7.3.4 and 7.4.1
	Object integration assistant	7.4.5
Legacy-Legacy integration	Transformation toolkit and assistant	7.3.4 and 7.4.1
	Schema and object integration assistants	7.4.5
Legacy-New comparison	Transformation toolkit and assistant	7.3.4 and 7.4.1
	Schema integration assistant	7.4.5
Mapping Definition	History	7.3.5
Wrapper Generation	History analyzer	7.5.1
	Wrapper encoders	7.5.2

Figure 7-2: Forward-reverse methodology processes and DB-MAIN tools.

7.3.1 Repository

The repository collects and maintains all the information related to the development of a federation. This usually comprises several schemas and texts. A *schema* is a description of the data structures to be processed, while a *text* is any textual material generated or analyzed (e.g. a program or a SQL script). The schemas of a project are linked through specific relationships.

The schema specification is based on a generic data model defined in Chapter 2. Besides the standard concepts 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.

7.3.2 GUI

User interaction uses a fairly standard GUI. Browsing through several sources requires an adequate presentation of the 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 7-3.

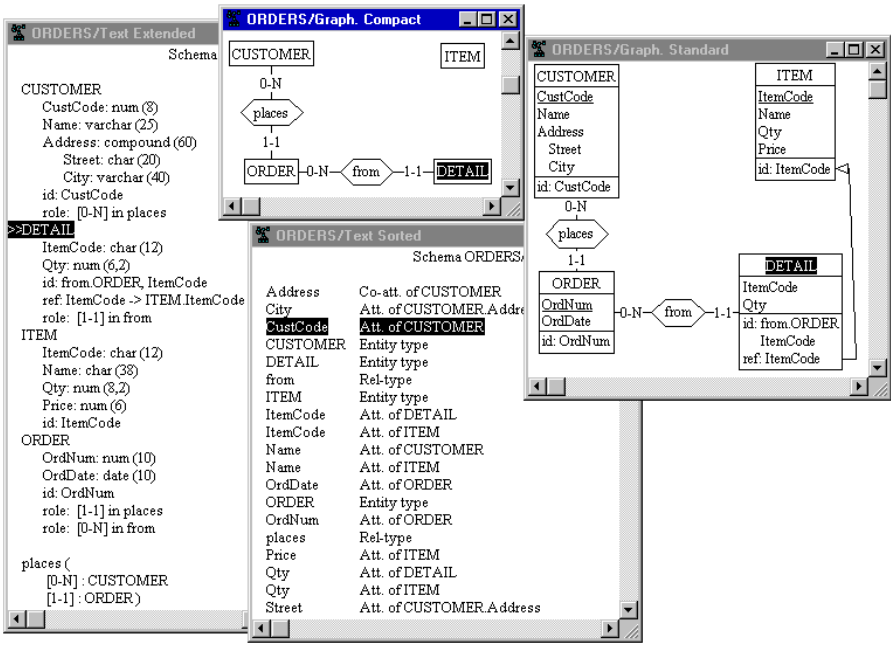


Figure 7-3: DB-MAIN can display a schema in six different formats. This screen copy shows four of them: text extended (left), text sorted (bottom), graphical compact (top) and graphical standard (right).

7.3.3 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 7-4 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
    sch:=GetCurrentSchema();                    /* what is the current opend schema? */
    if IsVoid(sch) then {                        /* 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 7-4: A Voyager 2 program example (from [Englebert, 2001]).

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 (Chapter 3) 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 7-5 illustrates the toolbar for the attribute transformation. The current version of DB-MAIN proposes a toolset of about 30 elementary transformations whose some of them have been presented in Chapter 3, Section 3.4.

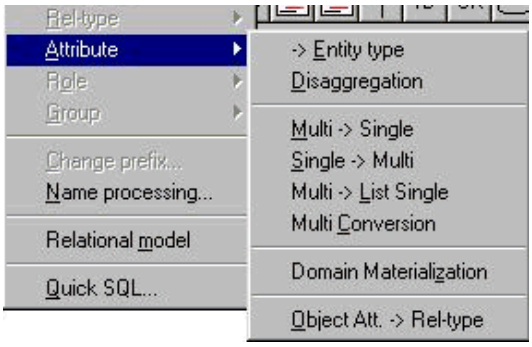


Figure 7-5: Transformation menu of an attribute. For instance, a selected attribute can be transformed into an entity type; a selected compound attribute can be disaggregated.

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 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 7.4.1.

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 with 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 3, Section 3.7. 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 History

DB-MAIN automatically generates and maintains a history log of all the schema transformation that are carried out when the developer derives a schema B from schema A. In Chapter 3, the history has been formalized in such a way that it can be replayed, analyzed and transformed. In practice, the history log file is a text file listing sequentially the signatures of all the transformations performed, with a well defined syntax. A complete log syntax has been developed in the DB-MAIN CASE environment. Note that this syntax is not of mathematical nature as in Chapter 3; rather, it uses an equivalent keyword based textual language.

Example

Figure 7-6 presents the 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"µ
*TRF att_to_et_inst
%BEG
        %NAM "address"
        %OWN 3 "SCHEMA"/"cobol-1"."Customer" 513
        %OID 523
        *POT "##1##getatt"
        *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 7-6: A history log example. The log records information about the transformation of the attribute address into an entity type by value representation.

The example of Figure 7-6 illustrates two essential features of the history:

- The history is made up of the formal signatures of the performed transformations stored sequentially in the exact order of performance.
- The history syntax (keywords, traditional notation conventions, structure, indentation) makes the history readable.

7.4 Methodological Assistants

An assistant is a higher-level solver dedicated to 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 five general purpose assistants which can support the processes of the forward-reverse methodology, namely, the *transformation* assistants (Section 7.4.1), the *schema analysis* assistant (Section 7.4.2), the *text analysis* assistant (Section 7.4.3), the *foreign key* assistant (Section 7.4.4) and the *schema integration* assistant (Section 7.4.5). These processors offer a collection of built-in functions that can be enriched by user-defined functions developed in *Voyager 2*.

7.4.1 Transformation Assistants

The *transformation assistant* (Figure 7-7) 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.

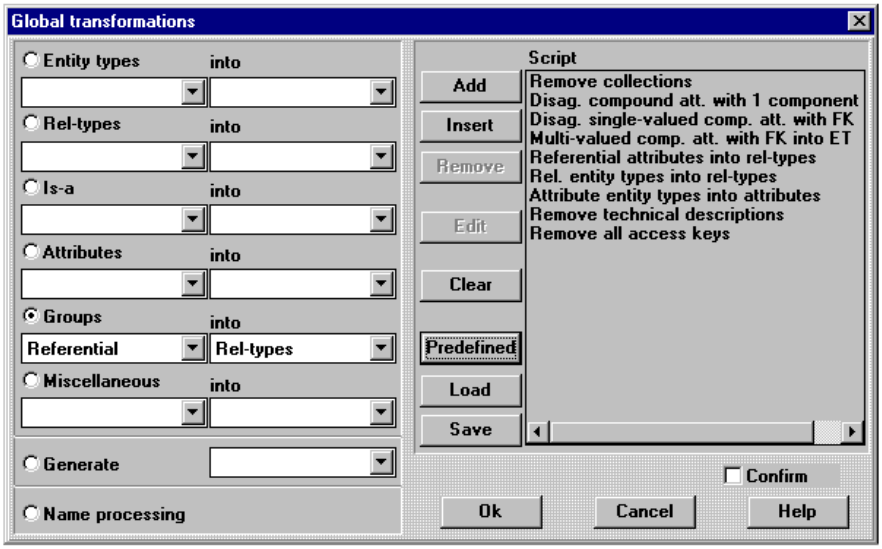


Figure 7-7: The basic global transformation assistant allows the analyst to perform a transformation on all the objects that satisfy a condition (left part). This screen copy shows that the analyst has developed a small script (right part) to conceptualize a COBOL schema. More complex scripts can be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

Moreover, the *advanced global transformations*, a sophisticated version of the *transformation assistant*, provides 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 7-8 presents the script of the transformation plan developed in Chapter 3, Section 3.7. We recall that the transformation plan has been defined for translating any schema expressed in the relational model into an equivalent schema expressed in the ER model.

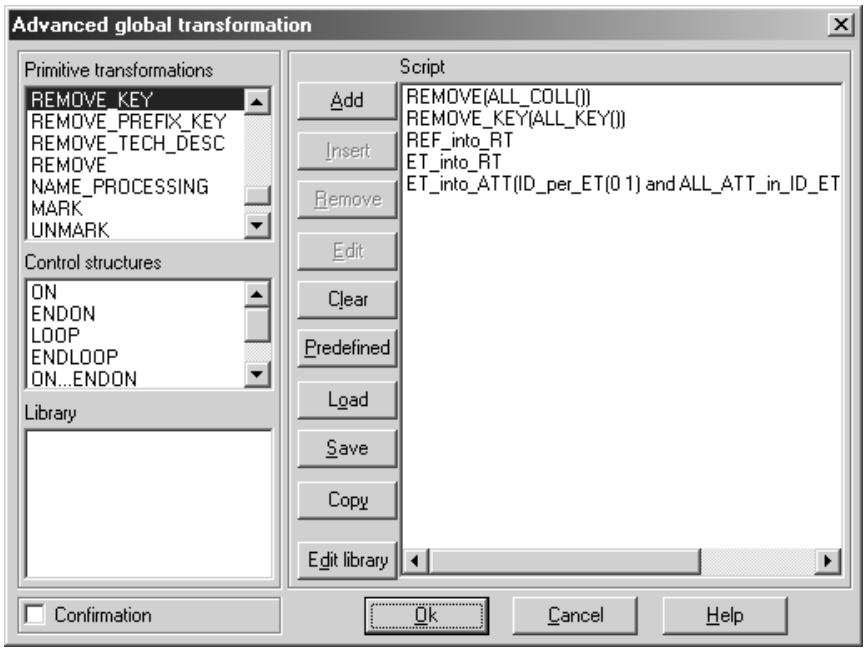


Figure 7-8: The advanced global transformation assistant allows the analyst to write a complex transformation plan such as the model translation between the relational model and the ER model.

7.4.2 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 logical 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, UML, 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 automate the detection of all the constructs and constraints that are not available in the wrapper logical model. Figure 7-9 presents the lists of rules that all the constructs must satisfy to comply with the wrapper logical model.

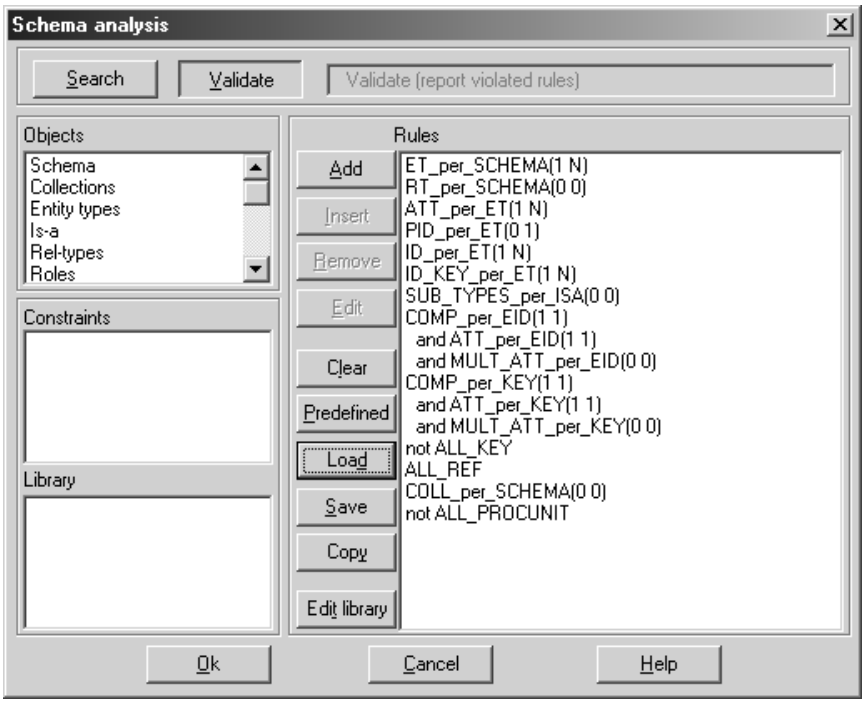


Figure 7-9: The schema assistant allows the analyst to detect the constructs that are allowed in a specific model.

7.4.3 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 four processors provided by this assistant:

- *Physical schema extractor.* The physical extraction process is carried out by a series of 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 [Weiser, 1984].

More detail on these processors can be found in [Henrard, 2003].

7.4.4 Foreign Key Assistant

The *foreign 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, the analyst gives the criteria to find the matching groups (name, structure, type, length, etc.). When the matching groups are found, he can create the foreign keys.

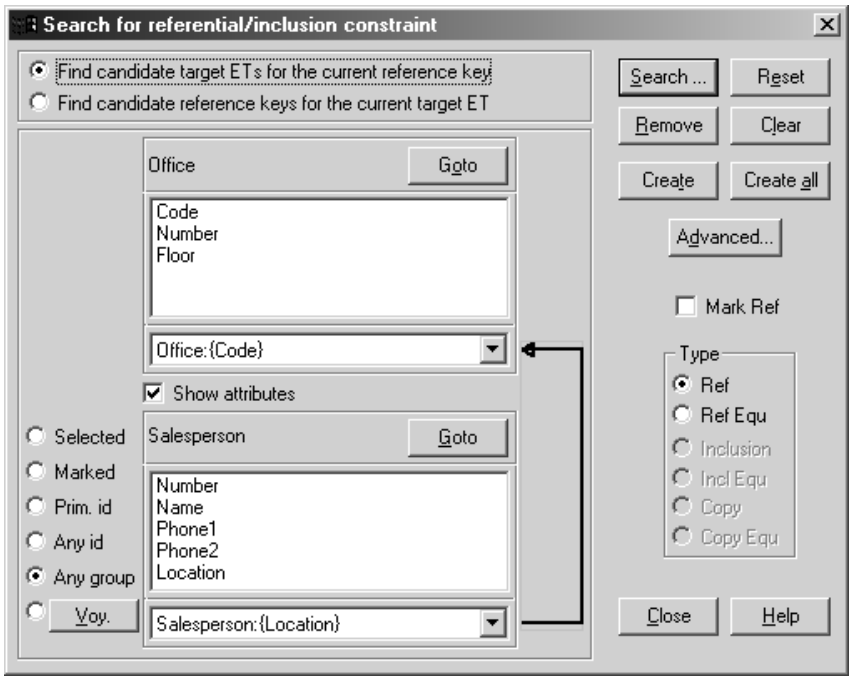


Figure 7-10: The dialog box of the foreign key searching assistant.

7.4.5 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 (Chapter 6, Section 6.2.2). 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.

Schema integration assistant

This assistant integrates a schema into another schema (Figure 7-11) by using predefined rules.

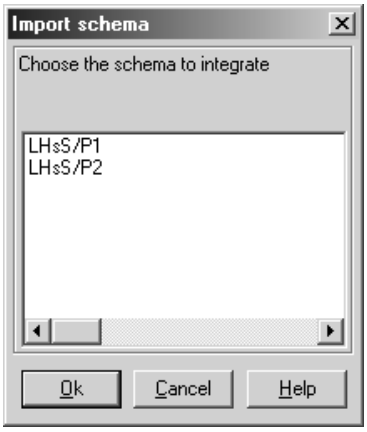


Figure 7-11: Schema integration assistant.

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:* 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:* 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:* 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 is 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:* add to the collection the entity types that were not connected.

Object integration assistant

The *object integration tool* (Figure 7-12) 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 7-13) 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 7-12 shows the integration of entity types Employee and Salesperson. When asserting that Employee.Number and Salesperson.Number are the same (Figure 7-12, button Same), the assistant compares their properties and presents them whenever a conflict is detected. Solving this conflict is up to the analyst (Figure 7-13).

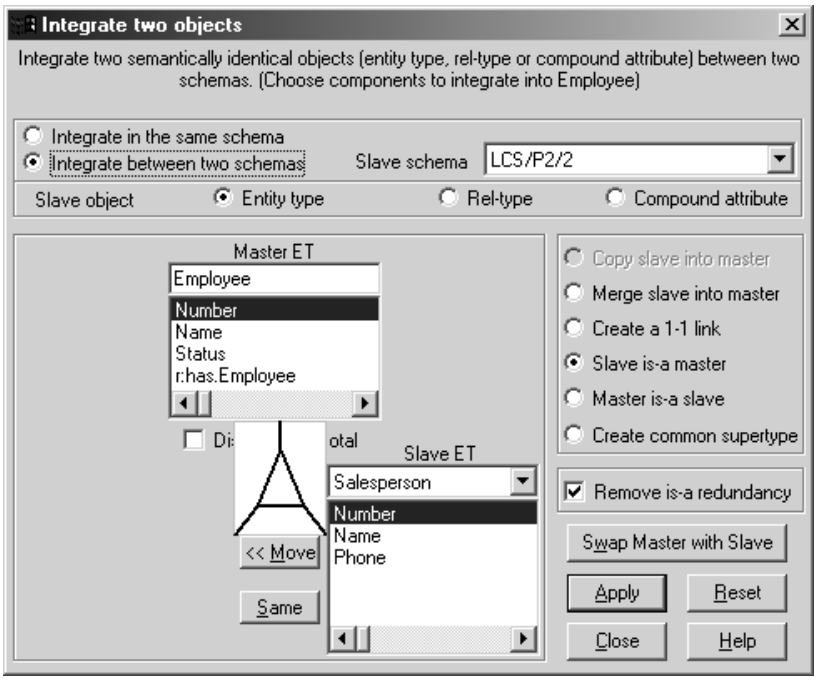


Figure 7-12: The integration assistant. Entity types Employee and Salesperson of two distinct local schemas are examined for integration. Among the six integration strategies, the analyst chose the fourth one, according to which Employee is a supertype for Salesperson. The attributes and roles are compared and either migrated (button <<Move) or merged (button Same). Here, the analyst is going to tell that attributes Employee.Number and Salesperson.Number have the same semantics, and that only the first one must be kept.

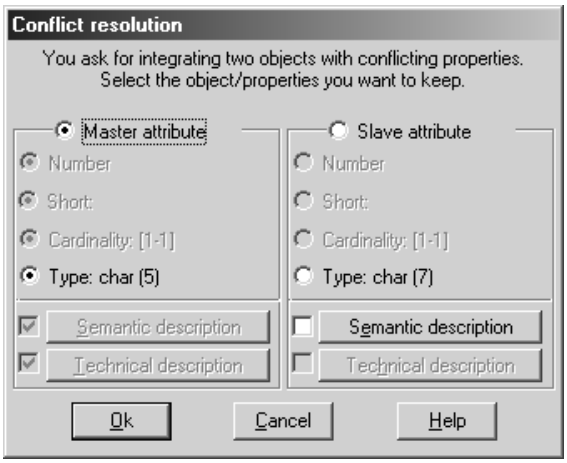


Figure 7-13: The integration assistant: resolving the conflicting properties of attributes Employee.Number and Salesperson.number that have been declared to be the same. Two conflicting properties have been identified: type and semantic description.

7.5 Wrapper Generation Tools

The InterDB tools have been built within the DB-MAIN environment (Figure 7-14). 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 7-14):

- *History analyzer.* The history analyzer analyses the history h in order to enrich the wrapper logical schema of logical/physical 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 InterDB logical wrapper (Chapter 5, Section 5.6.1) and a documentation for the programmers whereas the *object wrapper encoder* generates the Java code of the specific InterDB object wrapper (Chapter 5, Section 5.6.2) and the definition of the wrapper object schema

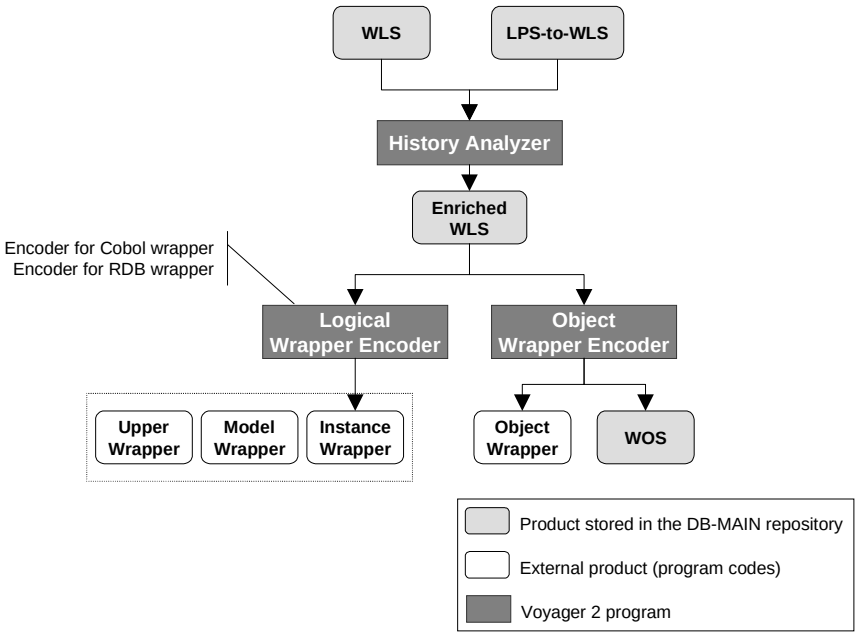


Figure 7-14: InterDB CASE tools: history analyzer and wrapper encoders.

All these tools are built around the DB-MAIN repository: 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.

7.5.1 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-to-WLS] and t=[lps-to-wls], then $\{h^{-1},t\}$ is the functional specification of the logical wrapper (Chapter 5, Section 5.5). Therefore, history h can be used to generate the wrapper. Moreover, since the history has a formal syntax, it can easily be interpreted by a program. 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 the minimal h in order to transform it into functional specifications from which the wrapper logical schema is enriched with physical/wrapper 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.

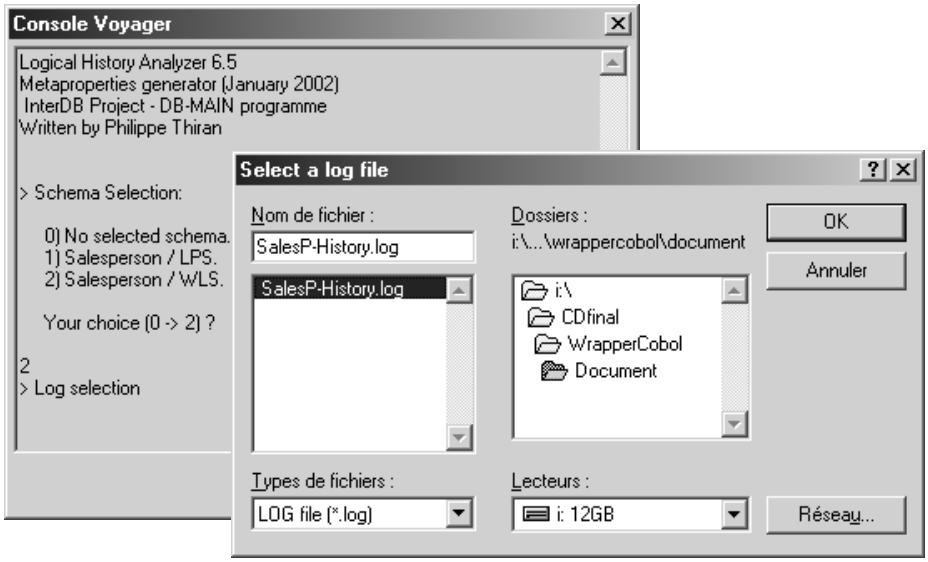


Figure 7-15: Voyager 2 console of the history analyzer. The analyst chooses the wrapper schema and the history log to be analyzed.

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/wrapper correspondence between a wrapper logical schema and the underlying physical schema.

The history analyzer proceeds in two main steps:

- It dynamically extends the DB-MAIN repository with meta-properties that represent the physical/logical correspondences. A meta-property is defined as a triple <name, construct, value domain> that specifies that a construct (construct) is associated with a meta-property (name) of value domain value domain. Some of these meta-properties are represented in Figure 7-16.
- It analyzes the history log that holds $\{h^{-1},t\}$ and computes, for each schema transformation, the meta-properties of the construct(s) associated to that schema transformation. Practically, it scans the history log according to predefined patterns that represent the schema transformation types taken into charge by the logical wrapper. Figure 7-17 illustrates the correspondences among schema transformation types, history patterns and meta-properties.

Meta-Property	Construct	Value
file-name	Entity type	to be defined by the user - only for COBOL
data-source-name	Schema	to be defined by the user - only for RDB
Index	Attribute	true | false
Down-mapping	Attribute Entity type	rename: name at the physical level
Down-mapping	Compound monovalued attribute	CompAtt-Single(*att*) | CompAtt-Serial(*list_att*)
Down-mapping	n-level attribute (n>1)	substring(offset,length)
Multi-mapping	Simple multivalued attribute	MultiAtt-Single(*att*) | MultiAtt-Serial(*list_att*)
opt-implementation	Optional attribute	to be defined by the user: user default / system default / null
default-str	Schema	to be defined by the user - string as the null value for the string values
default-num	Schema	to be defined by the user - number as the null value for the number values
FK-implementation	Group	declared / simulated: *declared* means explicit foreign key; *simulated* means implicit foreing key (that is emulated by the wrapper)
ID-implementation	Group	declared / simulated: *declared* means explicit identifier; *simulated* means implicit identifier (that is emulated by the wrapper)

Figure 7-16: Meta-properties defining the mapping properties between the physical and wrapper logical schemas.

Schema Transformation	History Pattern (Keyword)	Meta-Property
Rename-ET	MOD ENT *with* %NAM	<Entity type, Down-mapping, Rename>
Rename-Att	MOD ATT *with* %NAM	<Attribute, Down-mapping, Rename>
Serial-CompAtt	TRF AGREG	<Compound attribute, Down-mapping, CompAtt-Serial> <Attribute, Down-mapping, Rename>
Single-CompAtt	TRF MONO_TO_COMP	<Compound attribute, Down-mapping, CompAtt-Serial> <Attribute, Down-mapping, Rename> <Attribute, Down-mapping, Substring>

Single-MultAtt	TRF MONO_TO_MULT	<Multivalued attribute, Down-mapping, MultAtt-Single>*
Serial-MultAtt	TRF GR_TO_MULT	<Multivalued attribute, Down-mapping, MultAtt-Serial>
Create-Identifier	CRE GRP or MOD GRP *with* %FLA "P" *or* %FLA "S"	<Group, ID-Implementation, Simulated>
Create-Reference	CRE CST	<Group, FD-Implementation, Simulated>
Del-AccessKey	DEL GRP or MOD FRP *with* %FLA "K	<Attribute, Index, true>
Modify-Cardinality	MOD ATT *with* %CAR	<Attribute, Opt-implementation, *user*>

Figure 7-17: Types of schema transformation and their corresponding history patterns and meta-properties.

Example

As an illustration, let us assume the excerpt of the history log depicted in Figure 7-18. The history analyzer parses the history log. First, it extracts the type of the schema transformation and the involved constructs:

- the keyword MOD ENT means the modification of an entity type;
- the different values following the two keywords %NAME mean a renaming;
- the values after the keyword OLD ENT specify that the entity type in LPS has the name Person;
- the next values specify that the same entity type in WLS has the name Salesperson.

Then, it uses this information to set the meta-property of the entity type Salesperson of WLS: Down-Mapping: Rename, Person.

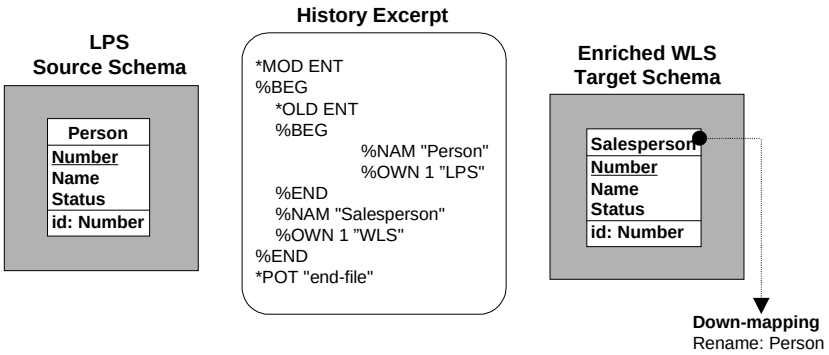


Figure 7-18: Example of the WLS enrichment.

7.5.2 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

From the enriched WLS, the *logical wrapper encoder*s produce the procedural code of the InterDB logical wrapper and a documentation for the programmer and the environment data that will allow the communication between the logical wrapper and the InterDB driver. These encoders are based on the wrapper development strategy developed in Chapter 5, Section 5.5. They produce the program code of the InterDB logical wrappers presented in Chapter 5, Section 5.6.1.

The logical wrapper encoder for COBOL DMS is fully documented in [Thiran, 2002e] whereas this for RDBMS is specified in [Chougrani, 2000].

Object wrapper encoder

The object wrapper encoder generates the interface definition for each object type of the wrapper object schema (WOS) and their implementation (Chapter 5, Section 5.6.2). 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. The object wrapper encoder performs two main tasks:

- *WOS definition within DB-MAN:* it defines the WOS from any WLS by applying a transformation plan between the logical data model and the object-oriented data model (Chapter 2, Section 2.4.2).

- *Program code generation:* for each object type in WOS, it generates its Java interface definition and its implementation.

The object wrapper encoder is fully documented in [Thiran, 2002g].

Chapter 8

Summary and Conclusions

In which the main ideas of this thesis are summarized and the directions for further research are suggested.

8.1 Summary

In this thesis, we have focused on the legacy and semantic aspects of database federation; that is, we have discussed modeling and architectural problems that arise when defining an integrated view of pre-existing and legacy databases. Our discussion of these problems has been characterized by the attention for issues in legacy information recovery and in the way in which legacy information can be reused and adapted to meet new requirements. We have thus strived to describe and use as much of the semantics of the legacy databases, to arrive at a global view that integrates both the actual requirements and the contribution of the database federation.

In this light, we have proposed a generic integration framework for expressing all the schemas and mappings of a database federation. This has been discussed in *Part I*. As legacy databases are typically based on different data models, in *Chapter 2*, we have presented a generic data model able to represent the schemas whatever their underlying data model and their abstraction level. In *Chapter 3*, we have formally defined the mappings as schema transformations on the generic data model. We have showed how these transformations can be used to automate the query translation between schemas.

The transformational paradigm is used as a rigorous formalism to define inter-schema mappings between the new requirements and the legacy information hold in the legacy databases. This issue is discussed in *Part II* of this thesis. In *Chapter 4*, we have presented a schema-

oriented framework for database federation that addresses four independent kinds of problems, including semantic enrichment and homogenization. We also have suggested that entrusted data wrappers with some responsibilities traditionally taken in charge by mediators can lead to significant improvement of the federation according to several criteria. The analysis has shown that system scalability, which is a critical issue in federated systems, can be better reached than through standard approaches. The implementation of this approach has been developed in Chapter 5, where the data wrapper technology is presented. We also have investigated their semi-automated development and showed that the schema-oriented approach ensures stronger opportunities of automating most of their software components.

Part III proposes a systematic forward-reverse methodology to build the schema-oriented framework for data mediation. In *Chapter 6*, we have not limited the integration methodology to legacy information but we have extended it to the new requirements. This way, we have developed a methodology that relies on reverse engineering, homogenization and transformational techniques to recover the semantics of the legacy databases and to identify the horizontal and vertical mappings between the legacy databases and the new requirements.

Like any complex process, building a database federation cannot be successful without the support of adequate tools called CASE tools. This part has been discussed in *Part IV*. A short analysis of the market showed that no current CASE tools can help neither in reverse engineering, nor in homogenizing and integrating complex databases. Hence the importance of the DB-MAIN CASE tool presented in *Chapter 7*. In particular, this thesis reports on two original aspects of DB-MAIN. First, DB-MAIN gives the analyst an integrated toolset for the development of database federation, providing, for instance, functions for reverse engineering, homogenization, integration and mapping definition. The second original aspect of DB-MAIN is its *Voyager 2* meta-language that has allowed us to develop repository-based analyzers and wrapper generators.

Hence the main contributions of this thesis in the building of interoperation frameworks that offer a virtual and integrated view of both legacy and new systems:

- We have investigated the role of *schema transformations* as the unifying basis of the software production and the forward-reverse methodology. At the software level, we have shown how schema transformations are used to automate the translation of queries. At the methodology level, we have shown how these schema transformation techniques can cope with the processes of the forward-backward methodology.
- We have developed a *wrapper development technology* for legacy databases. Interestingly, our wrappers provide primitives not only for data extraction, but also for data modification. Since the elicited constraints have been coded in the wrapper, newly developed programs can profit from automatic constraint management that the underlying DMS cannot ensure.
- We have proposed a *forward-reverse methodology* that enjoys the following properties:
    - *Scalability:* new levels and new databases can be incorporated with minimal cost. Including a new database in the federation involves modification that mainly is

taken in charge by the local sources.

- *Progressivity:* each step provides abstract functionalities that can be exploited immediately; this also ensures better development cost and time control, as well as better risk management.
- *Legacy preservation:* local databases and applications can be used while their functions meets the local needs.
- *Maximal reuse:* since the semantic structures of the legacy databases have been precisely elicited, the conceptual schema of new databases to develop includes the new requirements, and only them.
- *Simplicity of the integration process*: issues generally addressed in theoretical approaches to schema integration are of a lesser importance in our framework since they have been performed in the reverse engineering and homogenization processes.

8.2 Conclusions

One of the biggest challenges of the development of future information systems certainly is the reuse of valuable legacy components, and more particularly legacy databases. In essence, problems associated with such challenges are modeling and architectural problems. Hence, from a generic point of view, all that can be done, is to offer methods and techniques to support the solution of typical problems that arise in practice. This does not imply that once such techniques are available, the practical problems are easy to deal with. The reverse engineering of a database, for example, remains a difficult and tedious task even when DBRE techniques and supporting tools are available. Also, integrating local databases is a very difficult task, especially in the presence of semantic heterogeneity.

This does not mean that scientific research in this area is infeasible. Taking this particular research as an example, we feel that the schema transformation paradigm is an important baseline when constructing such systems in a real-life situation. This is especially true for the more complex application domains as those mentioned above. Moreover, many of our results have applicability in broader contexts as well. Examples include the data wrapper technology which can be used in multidatabase systems or migration projects, and the notion of the homogenization which is relevant to data warehouse systems.

8.3 Further Research

We feel that further research in this area should be *practical*. At the moment, we have proposed some insights into the semantic problems that arise when integrating legacy databases

into new systems and an architectural solution for wrapping them has been proposed. It is now time to find empirical support for these insights, and judge the feasibility of the approach in the real world. One of them is determining the optimal structure of the schema hierarchy of federated databases. The thesis suggests four criteria that still are to be evaluated in practice.

Another issue is the more specific problem of developing wrappers that control the concurrence. The challenge is to permit concurrent updates to the underlying legacy systems without violating their autonomy.

"Quand nous parvenons au but, nous croyons que le chemin a été le bon." - Paul Valéry

References

[Aslan, 1999] G. Aslan, D. McLeod, "Semantic Heterogeneity Resolution in Federated Databases by Metadata Implementation 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.

[Bancilhon, 1981] F. Bancilhon, A. Bonner, "Update Semantics of Relational Views", *ACM Trans. on Database Systems*, 6(4), pp. 557-575, 1981.

[Batini, 1986] C. Batini, M. Lenserini, 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.

[Blaha, 1998] M. Blaha, W. Premerlani, *"Object-Oriented Modeling and Design for Database Applications"*, Prentice Hall, 1998.

[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 1999] S. Busse, R-D. Kutsche, U. Leser, U. Leser, *"Federated Information Systems: Concepts, Terminology and Architectures"*, Technical Report, Technical University of Berlin, 1999.

[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.

[Cardenas, 1987] A. Cardnas, "Heterogeneous Distributed Database Management: the HD-DBMS", Proceedings of IEEE, 1987.

[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 *Procceding of CoopIS'98*, IEEE Computer Science Press, 1998.

[Chougrani, 2000] M. Chougrani, "Relational Wrapper Generator", *Research Report*, InterDB Project, University of Namur, 2002. In French.

[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.

[Conrad, 2002] S. Conrad, W. Hasselbring, A. James, D. Kambur, R-D. Kutsche, Ph. Thiran, "Report on the EFIS 2001 Workshop", *The Computer Journal*, vol. 45, 2002.

[Date, 1995] C.J. Date, *"An Introduction to Database Systems"*, 6th Edition, Addison-Wesley, 1995.

[Dayal, 1982] U. Dayal, P. Bernstein, "On the Correct Translation of Update Operations on Relational Views", in *ACM Trans. on Database Systems*, 8(2), pp. 381-416, 1982.

[Dayal, 1982b] U. Dayal, H-Y. Hwang, "View Definition and Generalization for Database Integration in Multibase: A System for Heterogeneous Distributed Databases", in *Proceedings of Berkeley Workshop*, pp. 203-238, 1982

[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, Releigh, 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.

[Dupont, 1996] Y. Dupont, "Resolving Fragmentation Conflicts in Schema Integration", in Proc. of *ER'94*, LNCS, pp. 513-532, Springer-Verlag, 1994.

[Elmagarmid, 1999] A. Elmagarmid, M. Rusinkiewicz, A. Sheth, *"Management of Heterogeneous and Autonomous Database Systems"*, Morgan Kaufmann, 1999.

[Elmasri, 1994] R. Elmasri, S. B. Navathe, *"Fundamentals of Database Systems"*, Addison-Wesley, 1994.

[Englebert, 2001] V. Englebert, *"Voyager II Manual"*, DB-MAIN Series, Institut d'Informatique, University of Namur, 2001.

[Gall, 1995] H. Gall, R. Klösh, "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, editors, 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.

[Garcia, 2000] H. Garcia-Molina, J.D. Ullman, J. Widom, *"Database System Implementation"*, Prentice Hall, New Jersey, 2000.

[Gardarin, 1999] G. Gardarin, F. Sha, T. Dang-Ngoc, "XML-based Components for Federating Multiple Heterogeneous Data Sources", in *Proccedings of ER*, LNCS, pp. 506-519, 1999.

[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. Gliglor and R. Popescu-Zeletin, "Transaction Management in Distributed and Heterogeneous Database Management Systems", *Information System*, 11(4), pp. 287-297, 1986.

[Graefe, 1993] G. Graefe, "Query Evaluation Techniques for Large Databases", in *ACM Computing Surveys*, 25(2), pp. 73-170, 1993.

[Gray, 1984] P. Gray, *"Logica Algebra and Database"*, John Wiley & Sons, New York, 1984.

[Gray, 1993] J. Gray, A. Reuter, *"Transaction Processing: Concepts and Techniques"*, Morgan Kaufmann Publishing, 1993.

[Hainaut, 1989] J-L. Hainaut, "A Generic Entity-Relationship Model", in *Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis*, North-Holland, 1989.

[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.

[Hainaut, 2002] J-L. Hainaut, *"Introduction to Database Reverse Engineering"*, Research Report, LIBD, University of Namur, 2002.

[Hainaut, 2002b] J-L. Hainaut, "UML and ER Comparison", Working Paper, LIBD Series, University of Namur, 2002.

[Hammer, 1995] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio and Y. Zhuge, "The Stanford Data Warehousing Project", *in IEEE Quart. Bull. Data Eng.*, 18(2), pp. 41-48, 1995.

[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.

[Henrard, 2002] J. Henrard, J-M. Hick, Ph. Thiran, J-L. Hainaut, "Strategies for Data Reengineering" in *Proceedings of WCRE*, IEEE Computer Society Press, pp. 211-220, 2002.

[Henrard, 2003] J. Henrard, *"Program Comprehension in Database Reverse Engineering"*, PhD Thesis, University of Namur, 2003.

[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.

[Hohenstein, 1996] U. Hohenstein, "Bridging the gab between C++ and relational databases", in *Proceeedings of tje Tenth European Conference on Object-Oriented Programming*, LNCS 1098, Springer-Verlag, pp. 398-420, 1996.

[Hull, 1986] R. Hull, "Relative Information Capacity of Simple Relational Database Schemas", *in SIAM Journal and Computing*, 15(3), pp. 856-886, 1986.

[Hull, 1997] R. Hull, "Managing Semantic Heterogeneity in Databases: A Theorical Perspective" in

Proc. of ACM PODS, 1997.

[Johnson, 1984] D.S. Johnson, A. Klug, "Optimizing Conjunctive Queries under Functional and Inclusion Depencies", *Journal of Computer and System Sciences*, 22(1), pp. 167-189, 1985.

[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.

[Kapitskaia, 1997] O. Kapitskaia et al. "Dealing with Discrepancies in Wrapper Functionality", INRIA Technical Report RR-138,1997.

[Kashyap, 1996] V. Kashyap, A. Sheth, "Semantic and Schematic Similarities between Database Objects: A Context-based Approach", in *VLDB Journal*, 54(4), pp. 276-304, 1996.

[Kashyap, 1997] V. Kashyap, *"Information Brokering Over Heterogeneous Digital Data: Metadatabased Approach"*, PhD Thesis, Rutgers, State University of New Jersey, 1997.

[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.

[Kim, 1990] W. Kim, *"Introduction to Object-Oriented Databases"*, MIT Press, Cambridge, 1990.

[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, "Mutltidatabase Interoperability", *IEEE Computer Magazine*, 19(12), pp.10-18, 1986.

[Litwin, 1994] W. Litwin, *"Multidatabase Systems"*, Prentice Hall: Englowood 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", *Reasearch Report*, INRIA Rocquencourt, 2001.

[Manolescu, 2001b] I. Manolescu, D. Florescu, D. Kossman, "Answering XML Queries over Heterogeneous Data Sources", *in Proceedings of the 27th VLDB Conference*, 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, 1998] P.J. McBrien and A. Poulovassilis, "A Formalisation of Semantic Schema Integration", *Information Systems*, 23(5), pp. 307-334, 1998.

[McBrien, 1999] P.J. McBrien and A. Poulovassilis, "Automatic Migration and Generation of Database Applications - a Schema Transformation Approach", in *Proceedings of ER'99*, LNCS 1728, Springer-Verlag, pp. 96-113, 1999.

[McBrien, 2000] P. McBrien, A. Poulovassilis, "Schema Evolution in Heterogeneous Database Architectures- A Schema Transformation Approach", *in Proceedings of CoopIS'00*, LNCS, Springer-Verlag, 2000.

[McBrien, 2002] P.J. McBrien and A. Poulovassilis, "Schema Evolution in Heterogeneous Database Architectures, A Schema Transformation Approach", In *Proceedings of CAiSE02*, LNCS 2348, Springer-Verlag, pp. 484-499, 2002.

[McBrien, 2003] P.J. McBrien and A. Poulovassilis, "Data Integration by Bi-Directional Schema Transformation Rules", in *Proceedings of ICDE03*, IEEE, 2003.

[Meng, 1995] W. Meng, C. Yu, "Query Processing in Multidatabase Systems", in W. Kim (editor) *Modern Database Systems*, Addison-Wesley, pp. 551-572, 1995.

[Miller, 1993] G. Miller, R. Beckwidth, C. Fellbaum, D. Gross and K. Miller, "Introduction to {WordNet}: An on-line lexical database", *Technical Report*, Princeton University, 1993.

[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.

[Navathe, 1996] S. Navathe, A. Savasere, "A Schema Integration Facility Using Object-Oriented Data Model", *in Object-Oriented Multidatabase Systems - a Solution for Advanced Applications*, Chapter 4, pp. 105-128, Prentice Hall, 1996.

[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.

[Potts, 1988] C. Potts, G. Bruns, "Recording the Reasons for Design Decisions", in *Proc. of ICSE*, IEEE, 1988.

[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 Metada Software Layer in Information Systems Integration", *in Proceedings of CAiSE'01*, pp. 299-314, LCNS 2068, 2001.

[Rodríguez, 2001] P. Rodriguez-Gianolli, J. Mylopoulos, "A Semantic Approach to XML-based Data Integration", in *ER Proceedings*, ", Springer-Verlag, 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.

[Rolland, 1993] C. Rolland, "Modeling the Requirements Engineering Process", in *Proc of the 3rd European-Japanese Seminar in Information Modeling and Knowledge Bases*, May 1993.

[Roland, 2003] D. Rolland, *"Database Engineering Process Modelling"*, PhD Thesis, Computer Sciences Department, University of Namur, 2003.

[Rosenthal, 1985] A. Rosenthal, D.S. Reiner, "Querying Relational Views of Networks", *Query Processing in Database Systems*, Springer-Verlag, 1985.

[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.

[Sattler, 2003] K-U. Sattler, S. Conrad, G. Saaker, *"Interactive Example-Driven Integration and Reconciliation for Accessing Database Federations"*, in *Information Systems*, 28, pp. 393-414, Elsevier, 2003.

[Savasere, 1991] A. Savasere, A.P. Sheth, G. Gala, S.B. Navathe, H. Markus, "On Applying Classification to Schema Integration", *in Proceedings of RIDE-IMS*, pp. 258-261, 1991.

[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.

[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, 1992] S. Spaccapietra, C. Parent and Y. Dupont, "Model Independent Assertions for Integration of Heterogeneous Schemas", *The VLDB Journal*, 1(1), pp. 81-126, 1992.

[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.

[Thieme, 1995] Ch. Thieme, *"Schema Integration Based on Structure and Behavior"*, PhD Thesis, University of Amsterdam, 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, 2000] Ph. Thiran, J-M. Hick, A. Chougrani, J-L. Hainaut, "CASE Support for the development of FIS", in *Proceedings of EFIS'00*, Infix, 2000.

[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*, IEEE Computer Society Press, 2001

[Thiran, 2001c] Ph. Thiran, J-L. Hainaut, "Evolving Hybrid Databases: Architecture and Methodology", in *Proceedings of EFIS'01*, Infix, October 2001.

[Thiran, 2002b] Ph. Thiran, "InterDB Driver - Logical Server and Java", *Research Report*, InterDB Project, University of Namur, 2002.

[Thiran, 2002c] Ph. Thiran, B. Noël, "Object Wrapper Server", *Research Report*, InterDB Project, University of Namur, 2002. In French.

[Thiran, 2002d] Ph. Thiran, S. Radulescu, "Reverse Engineering of Taxes Databases of the City of Namur", *Research Report*, InterDB Project, University of Namur, 2002. In French.

[Thiran, 2002e] Ph. Thiran, "Logical Wrapper: Wrapper for COBOL", *Research Report*, InterDB Project, University of Namur, 2002. In French.

[Thiran, 2002f] Ph. Thiran, "Logical Server and InterDB Driver" *Research Report*, InterDB Project, University of Namur, 2002. In French.

[Thiran, 2002g] Ph. Thiran, B. Noël, "Object Wrapper Encoder", *Research Report*, InterDB Project, University of Namur, 2002. In French.

[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 whith 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.

[van den Heuvel, 2002] W-J. van den Heuvel, *"Integrating Modern Business Applications with Legacy Systems"*, PhD Thesis, Tilburg University, 2002.

[van den Heuvel, 2002b] W-J. van den Heuvel, Ph. Thiran, "Constructing Federated Enterprise Schemas with Conceptualized Legacy Data Systems", *in Proceedings of WITS*, Barcelona, 2002.

[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.

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

[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. Fielt, "Scenarios for the Identification of Objects in Legacy Systems", in *Proceedings of WCRE'97*, IEEE Computer Society Press, 1997.

[Yan, 1997] L.L. Yan, M.T. Özsu, L. Liu, "Accessing Heterogeneous Data Through Homogenization and Integration Mediators", in *Proceedings of CoopIS'97*, IEEE Computer Sciences Press, 1997.

[Yan, 2001] L.L. Yan, R.J. Miller, L.M. Haas, R.Fagin, "Data-Driven Understanding and Refinement of Schema Mapping", in *Proc. of SIGMOD 2001*, 2001.