Wrapper-Based Evolution of Legacy Information Systems

PHILIPPE THIRAN and JEAN-LUC HAINAUT
Facultés Universitaires Notre-Dame de la Paix, Namur
GEERT-JAN HOUBEN
Vrije Universiteit Brussel
and
DJAMAL BENSLIMANE
Université Claude Bernard, Lyon

System evolution most often implies the integration of legacy components, such as databases, with newly developed ones, leading to mixed architectures that suffer from severe heterogeneity problems. For instance, incorporating a new program in a legacy database application can create an integrity mismatch, since the database model and the program data view can be quite different (e.g. standard file model versus OO model). In addition, neither the legacy DBMS (too weak to address integrity issues correctly) nor the new program (that relies on data server responsibility) correctly cope with data integrity management. The component that can reconciliate these mismatched subsystems is the *R/W wrapper*, which allows any client program to read, but also to update the legacy data, while controlling the integrity constraints that are ignored by the legacy DBMS.

This article describes a generic, technology-independent, R/W wrapper architecture, a methodology for specifying them in a disciplined way, and a CASE tool for generating most of the corresponding code.

The key concept is that of *implicit construct*, which is a structure or a constraint that has not been declared in the database, but which is controlled by the legacy application code. The implicit constructs are elicited through reverse engineering techniques, and then translated into validation code in the wrapper. For instance, a wrapper can be generated for a collection of COBOL files in order to allow external programs to access them through a relational, object-oriented or XML interface, while offering referential integrity control. The methodology is based on a transformational approach that provides a formal way to build the wrapper schema and to specify inter-schema mappings.

Authors' addresses: Ph. Thiran, Department of Business Administration, Facultés Universitaires Notre-Dame de la Paix, Rempart de la Vierge 8, 5000 Namur, Belgium; email: pthiran@fundp.ac.be; J.-L. Hainaut, Institute of Informatics, Facultés Universitaires Notre-Dame de la Paix, 21, rue Grandgagnage, 5000 Namur, Belgium; email: jlh@info.fundp.ac.be; G.-J. Houben, Department of Computer Science, Vrije Universiteit Brussel, Pleinlaan 2, 1050 Brussels, Belgium; email: gjhouben@vub.ac.be; D. Benslimane, IUT A Informatique, Université Claude Bernard, Lyon 1, Bâtiment Nautibus, 8, boulevard Niels Bohr, 96922 Villeurbanne cedex, France; email: djamal.benslimane@liris.cnrs.fr.

Categories and Subject Descriptors: D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Restructuring, reverse engineering, and reengineering*; D.2.12 [**Software Engineering**]: Interoperability—*Data mapping*; H.2.5 [**Database Management**]: Heterogeneous Databases—*Data translation*

General Terms: Design, Management, Reliability

Additional Key Words and Phrases: Schema transformation, data reverse-engineering, CASE tool, wrapper, legacy database, data consistency, evolution

1. INTRODUCTION

Information system evolution and migration are clearly among the most challenging engineering processes to date. Besides the complexity of making different technologies coexist in the same system, the architectural issues have become a major problem. Indeed the shift from centralized to distributed and cooperative paradigms has made the structure of legacy applications much more difficult to evolve than one or two decades ago [Ross 1993]. In particular the functional architecture of many of these applications is no longer fitted to the capabilities of modern technologies. That is what we will show for data consistency.

1.1 Semantic and Technology Mismatches

Legacy data-intensive applications were developed for data management systems that offered little help as far as data integrity was concerned. File management systems, early relational systems, hierarchical and even network data systems (despite their richness in data structuring) were notably insufficient to guarantee the level of data consistency required by modern client applications. As a consequence, consistency management has long been the responsibility of the client application code, an approach that resulted in the insertion in this code of data validation and consistency management sections. The code related to integrity constraints or data structures that were not explicitly taken care of by the data management system (what we will call *implicit* constraints and structures) had to be scattered in hundreds to thousands of places throughout the application code. These sections were written by different developers, with different skills, and in different styles and languages.

For example, in a legacy application based on standard files, the field `PR` of record type `ORDER`, that is used to reference a definite `PRODUCT` record, clearly is an *implicit foreign key*. Since file management systems ignore referential integrity, the latter must be ensured procedurally. In particular, before storing an `ORDER` record, a program must check the existence of a `PRODUCT` record identified by the value of field `PR`.

Plugging a new application component into such a legacy architecture often implies developing it according to these obsolete practices, notably by once again writing the validation code for the implicit data properties ignored by the data management system. Considering the example above: any new application program working on records of types `ORDER` and `PRODUCT` is forced to include referential integrity validation sections.

At the opposite end, reusing a legacy component and integrating it in a modern system poses similar problems. For instance, developing a new application on a legacy, but quite efficient, database creates an important semantic mismatch. Indeed, many important constraints and structures are ignored by the data management system, which generally offers semantically poor data structures (even modern RDBMS for instance still offer little more than *uniqueness* and *referential* built-in constraints). On the other hand, modern coding practices such as aspect-oriented programming naturally ignore these constraints as well. Indeed, data integrity codes, which were formerly included in the application code, are now supposed to have moved from the client application to the data server.

One of the most illustrative examples is the way referential integrity has been coped with before and now. In early data management systems, such constraints were either ignored (file systems, or Oracle up to V6 for example), or translated into simple and strongly limited one-to-many links (TOTAL, CODASYL and IMS DBMS for example). Therefore, the databases developed one or two decades ago generally include, whatever their technologies, thousands of implicit, undeclared, foreign keys, the management of which was ensured by the application code, as discussed above. When redeveloping such databases with modern relational technology, all these constraints can be explicitly declared or procedurally controlled (through triggers for instance), and therefore centrally managed by the DBMS itself. Clearly, when we couple a newly developed application component with a legacy database, the referential integrity is no longer ensured, neither by the DBMS nor by the client application.

1.2 Data Wrappers

In such a context, the problem of deficiencies in data integrity management can be addressed by dedicated components inserted between the legacy database and the application component, namely *wrappers*.

A wrapper is often used to extend the lifetime of components of existing data systems by facilitating their integration into modern distributed systems. In general, using data wrapping is an attractive strategy for several reasons. First, it does not alter the host system of the database. Second, it addresses the challenge of database heterogeneity by providing a standard and common interface. Third, it allows developers and data administrators to transparently incorporate new capabilities, such as statistics collection, performance monitoring, or integrity control. Finally, it provides a smooth path to a step-by-step modernization of a complex legacy system [Brodie and Stonebraker 1995].

The interface provided by a wrapper is made up of: (1) a wrapper schema of the wrapped database, expressed in some abstract canonical[1] data model and (2) a common query language for the data as viewed through the wrapper schema. Queries/updates on the wrapper schema are also known as *wrapper queries*/*updates*, while native ones directly performed on the legacy schema will be called *database queries*/*updates*.

[1]The term *canonical* is borrowed from the domain of Federated databases [Bouguettaya et al. 1998], to which this technology is strongly related.

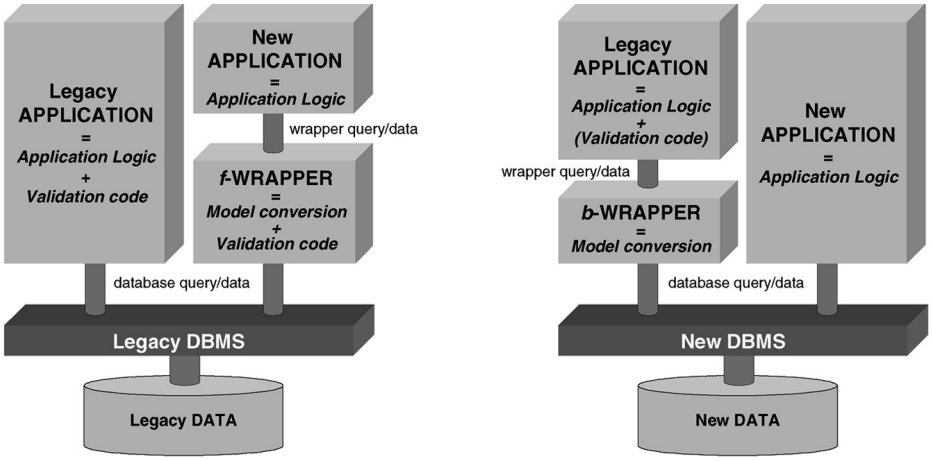


Fig. 1. The role of wrappers in the coexistence of legacy and new components in system evolution (left) and system migration (right).

In this article, we focus on database wrappers that both query and update the actual data, and in which the issue of data integrity control is of major importance. Such wrappers are sometimes called R/W *wrappers*, a term that we will use in this article.

1.3 Wrapper-Based Strategies for Information Systems Evolution and Migration

Though the reasoning we develop can apply to different resources, provided they can be wrapped, we will focus on the data component of information systems. Among the disruptions that can threaten the integrity of the data, *system evolution* due to technology change, and *system migration* certainly are the most demanding. Both can be made easier by integrating appropriate wrappers in the system architecture. We will discuss the role of data wrappers through the scenarios depicted in Figure 1.

The first scenario (Figure 1, left) is that of an evolving information system. It shows how a new application component can be added to a legacy system. Both legacy database and code are kept unchanged, while the new component is designed and developed following modern practices. In particular, (1) it is given a clean view of the data, devoid of any idiosyncrasy, such as redundancy and unnormalized structures, inherited from the legacy structures, and (2) it is not concerned with data integrity, which, according to current approaches, is the exclusive responsibility of the data server. In this context, the first role of the wrapper is to translate data and queries from the legacy data model and interface to those expected by the new component. The second role is to take charge of the validation logic that ensures data integrity. We will call such wrappers *forward wrappers* (*f-Wrappers* in Figure 1) since they emulate the new technology based on the legacy one. They will be the subject of this article.

The second scenario (Figure 1, right) supports the complex process of system migration, especially according to the *database first* approach [Wu et al.

1997]. The database is migrated first, preferably following a semantic procedure that consists in translating the semantics of the legacy database into the new technology.[2] The new components are developed on this new database, while the legacy components are interfaced, through wrappers, in order to allow them to read and write the new data. Since data integrity is ensured by both the new DBMS and the legacy applications,[3] the wrapper does not need to take care of this validation, so that its primary responsibility is model conversion. Such wrappers will be called *backward wrappers* (*b-Wrappers* in Figure 1), since they emulate the legacy technology on top of the new one. They are simpler than forward wrappers. Since backward wrappers do not support data consistency, they will be ignored in this article. The second scenario has been studied in Henrard et al. [2002], where various strategies for migrating structures, data and programs have been proposed and discussed.

In this article, we will study important architectural and design issues of forward R/W wrappers as far as data consistency management is concerned.

1.4 Bridging the Semantic Gap with Wrappers

The context stated so far is that of a new application component that updates the contents of a legacy database.

In the source system, that includes the legacy data and programs, the data are organized according to a set of constructs (structures and constraints) that can be classified in two categories. The first ones have been explicitly declared through the DDL code of the database schema, and are controlled by the legacy data management system. The schema that is made up of the *explicit* constructs will be called the *physical schema*. The other constructs are managed through different techniques external to the database, mainly validation sections in the application code and in the user interface. These constructs are called *implicit* from the database viewpoint. The *logical schema* of the database comprises both the explicit and implicit constructs. The new application components are supposed to access the data through their logical schema, and no longer include validation code for implicit constructs. As we have shown, the latter code will form the core of the forward wrapper.

To summarize, it is the responsibility of the wrapper to guarantee legacy data consistency by rejecting updates that violate implicit constraints, so that both legacy applications and new applications that update the data through the wrapper can coexist without threatening data integrity (Figure 1, left).

---

[2]As opposed to the *physical approach*, which consists in mapping physical structures of the legacy database in the closest physical structure of the new technology. For instance, a COBOL record type is converted into a relational flat table, and each top-level field into a column. Such an approach is fast and inexpensive, but results in poor database structures. This approach will be ignored in this discussion. See Henrard et al. [2002] for further details.

[3]This redundancy in data validation is a common problem in evolving systems that include components with different ages. Actually, it mainly poses a performance problem, since possibly expensive validation can be carried out twice. Its best known incarnation was the migration of Oracle V5/V6 databases to V7, which was the first version to effectively manage referential integrity. The experts suggested (1) to upgrade the schema by declaring foreign keys and (2) to deactivate their validation when legacy applications were running.

For example, the logical schema of a database that comprises a collection of coordinated COBOL files will include unique keys and foreign keys. In general, the former are *explicitly* translated into record keys and alternate record keys, while the referential integrity implied by the foreign keys are *implicitly* managed by the application code. New application components will rely on the forward wrapper to ensure both referential and (transitively) uniqueness integrity.

The process through which the implicit constructs can be identified and the full logical schema can be recovered is called *Database reverse engineering* [Aiken 1996]. Its scope generally is wider than is strictly needed in system evolution (the goal of the whole process is to recover the *conceptual schema* of the legacy database). However, its first step, namely recovering the logical schema, as well as the physical-to-logical mappings, is precisely what is required to develop forward R/W wrappers.

1.5 Current Approaches

Several prototype wrapper systems for legacy databases have been developed. They share two common characteristics against which we can compare our approach, namely the level of transparency and the update facilities they provide:

1. *Wrapper transparency.* Several approaches: Lim and Lee [1999], Roth and Schwarz [1997], and more recently those intended to produce XML views of relational schemas Bergamaschi et al. [2001], Carey et al. [2000], Fernandez et al. [2000], or Shanmugasundaram et al. [2001] consider a wrapper as a pure model converter, a software component that translates data and queries from the legacy DBMS model, to another, more abstract and DBMS-independent model. That is, the wrapper is only used to overcome the data model and query heterogeneity in database systems, leaving aside the added value that can be expected from wrapping. In such approaches, the semantic contents of both database and wrapper schemas are identical: the wrapper schema just translates explicit constructs and ignores implicit ones.
2. *Update facilities.* From the application point of view, it is natural that an update request expressed on the wrapper schema be automatically mapped onto the underlying legacy database. However, very few of the current wrappers supply this kind of support (e.g., Carey et al. [2000] or Fernandez et al. [2000] provide support for insertable and updateable XML views). Additionally, wrappers that support updates generally do not consider implicit constraints: they only provide update query mappings without any data consistency verification. In Carey et al. [2000] or Fernandez et al. [2000], only the (explicit) constraints declared in the relational database are controlled by the DBMS.

1.6 Objectives

In previous references [Thiran and Hainaut 2001; Thiran et al. 2005a], we described a general architecture for read-only wrappers, in which we addressed the query translation function. We also stated the principles of a methodology

for developing such wrappers and described briefly a CASE tool that supports it. The architecture and the methodology are both based on the *transformational* paradigm that provides a rigorous framework for automatically translating queries and assembling the requested data.

In the present article, we extend these results by addressing the problem of *update translation* in forward wrappers that include the control of implicit constructs of *legacy databases*. We consider wrappers that export a wrapper schema augmented with integrity constraints and structures that are not defined in the database schema. Updating data through such a wrapper poses the problem of guaranteeing legacy data consistency by rejecting updates that violate all the constraints, whether they are implicit or explicit.

This article partly relies on the contents of previously published references (notably Thiran et al. [2004], of which it is an extended version). It has been made self-contained in such a way that only readers interested in more in-depth detailed development will need to consult them.

This article is organized as follows. Section 2 develops a small case study that allows us to identify some of the main problems to be solved. Section 3 presents the main aspects of the architecture of wrappers that guarantee legacy data consistency with respect to both explicit and implicit constraints. Section 4 presents our schema transformation framework for specifying query and update mappings as well as implicit constraints. Section 5 deals with developing wrappers for legacy databases in a semiautomated way. The generation is supported by an operational CASE-tool, namely DB-MAIN. Section 6 presents some metrics of the wrapper development cost. They illustrate the necessity of wrapper development or construction. Finally, Section 7 concludes this article.

2. BUILDING R/W WRAPPERS—A FIRST INTUITIVE APPROACH

In this section, we develop a small example that illustrates some of the problems we intend to address in this article. We consider the sales administration of a company, which is based on a legacy relational DBMS, such as Oracle V5, in which no primary or foreign keys could be declared.

New economic trends force the company to evolve its information system. The company decides to keep its database unchanged but to build a wrapper that must allow new applications to retrieve and update the sales data. Legacy local applications are preserved while new ones can be developed through a safe interface. This safe interface is ensured by the wrapper, which guarantees the integrity and quality of the data flowing between the database and the new applications.

This objective raises the critical problem of developing a wrapper that allows updates that satisfy a set of constraints that can be either explicit or implicit. In particular, the wrapper should be able to make explicit, and manage, hidden constraints such as foreign and primary keys that are absent from the legacy physical model. This allows the behavior of the local applications that access the legacy database to be preserved.

The main problems in building such a wrapper are to define the wrapper schema from the database schema and to define the mappings between them.

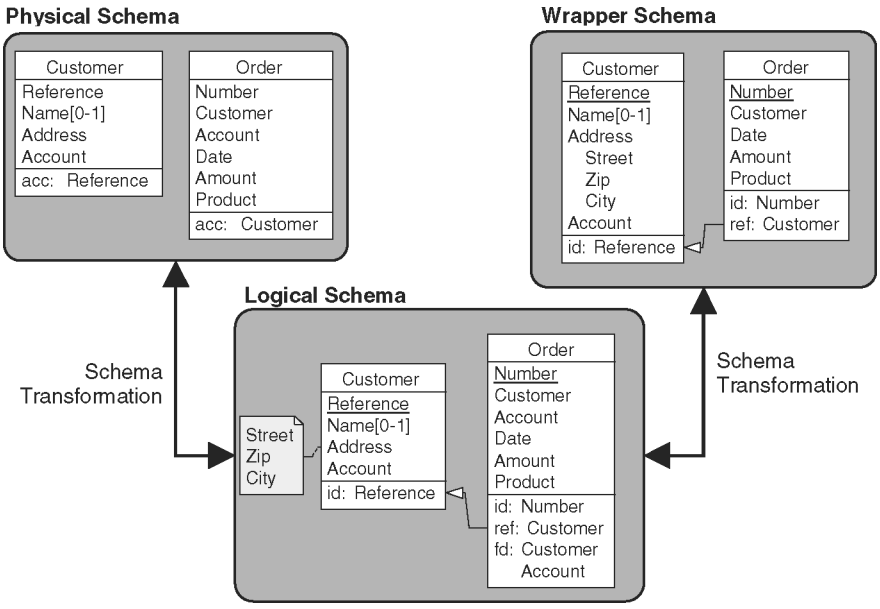


Fig. 2. The physical database, logical and wrapper schemas.

For practical reasons, we express the wrapper schema in a canonical model that is compliant with both physical and logical legacy models such as standard files, SQL-2, hierarchical, and network models. However, the nature of this model is not relevant to the architecture we are discussing (see Hainaut [2005] for more detail).

2.1 Wrapper Schema Definition

By analyzing the SQL DDL code, we can extract the database physical schema of the legacy database (Figure 2). The physical schema comprises the tables `Customer` and `Order`. Optional (nullable) columns are represented by the `[0-1]` cardinality constraint and indexes by the `acc`(ess key) constructs.

This process is fairly straightforward since it is based on the analysis of declarative code fragments or data dictionary contents. However, it recovers explicit constraints only, ignoring all the implicit constraints that may be buried in the procedural code of the programs. Hence the need for a refinement process that cleans the physical schema and enriches it with implicit constraints, providing the logical schema. Their elicitation is carried out through reverse engineering techniques such as program analysis and data analysis [Hainaut 2002], that are illustrated below:

1. *Program analysis:* before inserting a new order, the local applications check whether the customer number is already recorded in the database (implicit foreign key).
1. *Data analysis:* if `Reference` is a primary key of `Customer`, then its values must be unique, a property that will be checked through a query such as the

following:

```
select * from Customer
group by Reference
having count(Reference)> 1
```

The data analysis can also find hints that suggest the presence of a redundant attribute `Account` expressed by a functional dependency: `Customer` → `Account`. We therefore obtain the logical schema shown in Figure 2. New constraints now appear, such as primary keys (`id` constructs), foreign keys (`ref` constructs), and a functional dependency: `fd: Customer` → `Account`.

The next phase consists in interpreting and exporting the logical schema, therefore providing the wrapper schema through which the new client applications will view the legacy data. The logical schema expressed in an abstract data model must comply with the operational data model of the wrapper. In addition, this schema still includes undesirable constructs, such as redundancies and other idiosyncrasies, which must be managed but also hidden from the client applications, and therefore discarded from the wrapper schema.

The logical schema of Figure 2 includes a property stating that `Address` is often split into three pertinent fragments. Moreover, it depicts a structural redundancy: the attribute `Account` of `Order` is a copy of the attribute `Account` of `Customer`. To hide this redundancy, the attribute `Account` of `Order` does not appear anymore in the wrapper schema.

2.2 Mapping Definition

Once the wrapper schema has been built, we have to state how wrapper retrieval and update queries can be mapped onto legacy data. In the schema hierarchy mentioned above, the transitions between physical and wrapper schemas can be expressed as formal transformations on structures (such as discarding, renaming or aggregating), and on constraints (such as adding primary keys, foreign keys and functional dependency). The complexity of the transformation depends on the distance between the logical and wrapper schemas. For instance, an XML-based wrapper schema would require more sophisticated mapping rules than those mentioned above [Thiran et al. 2005b].

The database/wrapper mappings can then be built by interpreting the transformations on structures as two-way data conversion functions whereas the implicit constraint management can be emulated by transformations on constraints. For instance, let us assume the wrapper update shown in Figure 3, which inserts an instance of Order. By analyzing the update statement, the wrapper dynamically generates a sequence of operations that emulate and manage the implicit structures and constraints.

As exhibited through references (1) to (5) in Figure 3, the main operations carried out by an R/W wrapper when executing an update, are the following:

1. *Implicit constraint management:* the wrapper checks the satisfaction of the constraints implied by the implicit identifier (1) and the implicit foreign key (2).

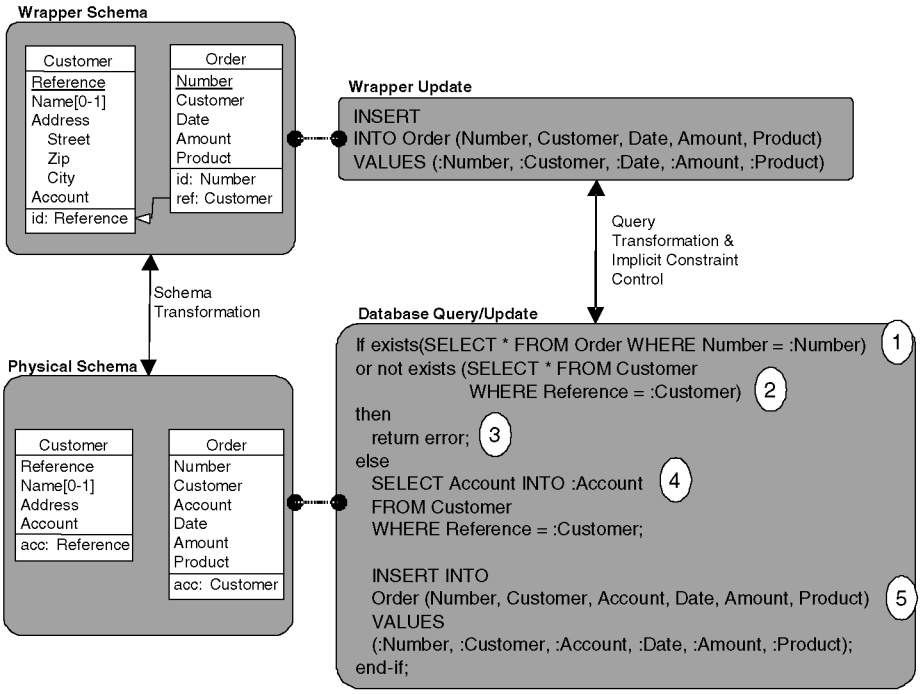


Fig. 3. Example of update translation and implicit constraint management.

2. *Data error management:* the wrapper reports possible implicit constraint violation (3).
3. *Redundancy management:* the wrapper controls the redundant attributes by assigning the value it gets from the source data (4).
4. *Query translation:* translation of the wrapper update against the wrapper schema into updates on the physical database schema (5).

3. WRAPPER ARCHITECTURE

3.1 General Architecture of an R/W Wrapper

In this section, we develop a generic architecture for R/W wrappers that provides both extraction and update facilities and that controls the implicit constructs of the source databases. This leads these wrappers to emulate advanced services such as integrity control and transaction and failure management, if the underlying DBMS does not support them.

The functionalities of such a wrapper are classified into functional services (Figure 4), among which we mention those that are relevant to the update aspects:

1. *Query / update analysis.* The wrapper query is first analyzed so that incorrect updates are detected and rejected as early as possible.

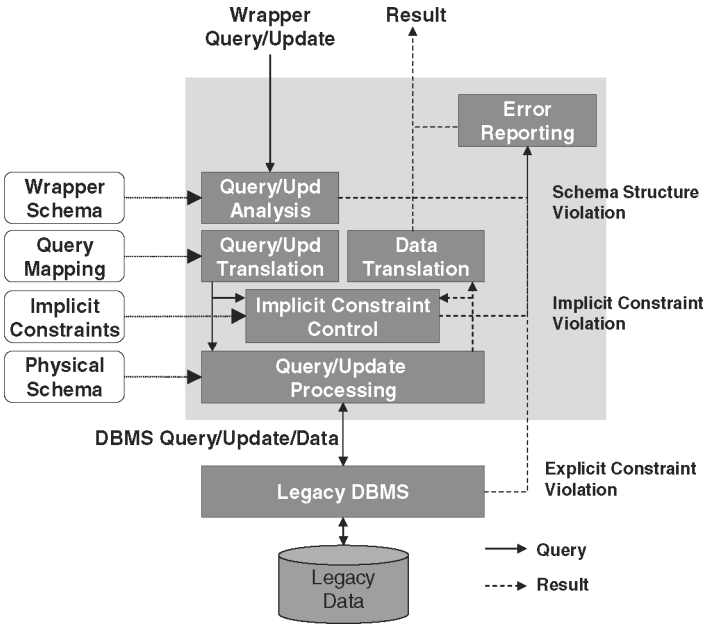


Fig. 4. R/W wrapper architecture.

2. *Error reporting.* The wrapper reports errors back to the client application.
3. *Query/update and data translation.* This refers to operations that convert data and queries/updates from one model to another.
4. *Implicit constraint control.* The wrapper emulates the implicit integrity constraints.
5. *Security.* The wrapper protects the data against unauthorized access and accident that may affect the integrity of the data.
6. *Concurrency and transaction management.* This function controls concurrent updates of the underlying legacy databases. This includes transaction and failure management.

The latter two services will be ignored in this article, in which we only develop and discuss the first four classes of functionalities (see Lawrence et al. [1998] for concurrency management, and Souder and Mancoridis [2000] for security management).

3.2 Wrapper Query/Update Analysis

Query analysis enables rejection of queries for which further processing is either impossible or unnecessary. The main reason for rejection is that the query is syntactically or semantically incorrect, which means that it refers to attributes or entity types that are undefined in the wrapper schema, or which have the wrong type. When one of these cases is detected, a diagnostic is returned to the user (see Section 3.3). Otherwise, query processing goes on.

3.3 Error Reporting

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

1. *At the legacy DBMS level:* legacy DBMS return some indicators on completion of query execution.
2. *At the wrapper level:* the wrapper catches internal errors. For example, it detects query syntax errors (see Section 3.2) or violation of implicit constraints (see Section 3.5).

In addition to the error codes a wrapper detects, it must provide standardized error codes of DBMS-specific errors to give new applications a standard way of dealing with error conditions. Although DBMS of the same family return similar kinds of errors, each does it in a different way, through different error numbers, message types, programming styles (return code, exception, triggered procedure). 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 (DBMS 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.

3.4 Query/Update and Data Translation

Query translation is the core function of a wrapper. It refers to operations that translate queries between two schemas (the database and wrapper schemas) and two languages (the database and wrapper query languages).

3.4.1 *Correctness and Efficiency.* The main function of the translation service is to transform wrapper queries into queries understandable by the underlying DBMS. Such transformation must ensure both correctness and efficiency:

—*Correctness.* This issue is addressed by the transformational paradigm. Since the mappings are based on reversible transformations that are proved to be correct, all the wrappers that are built according this approach can be considered correct. In Section 4, we describe the formal framework of these reversible transformations, which allows the 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. The same wrapper can lead to several execution strategies according to the way queries are processed by the underlying legacy DBMS. The access plan, optimization, and processing of the wrapper queries must use the legacy query capability and optimization.

3.4.2 *Query Translation Principles.* The translation service relies on the use of schema transformations that provide mechanisms for formally defining schema correspondence between the database and wrapper schemas. By

replacing the schema construct names in the wrapper query with their mapping expression in terms of database schema, we produce a database query that directly addresses the actual data.

As is typical in multilanguage translation, wrapper queries are first converted into an internal pivot language. Query transformations are applied, and then the result is translated into the DBMS language. The pivot language, which is based on a generic data model (Section 4), is independent of DBMS query and manipulation languages. We can now state the three main successive steps of query translation:

—*Wrapper query mappings:* syntactic translation of the wrapper query into the internal pivot language;
—*Inter-schema mappings:* translation of the internal query following the schema transformation between the database and the wrapper schemas;
—*Language mappings and optimization:* translation of the internal query in the DBMS query language. Producing an efficient execution strategy depends on the syntax and expressiveness of both the wrapper (or internal) and DBMS processing capabilities.

3.4.3 *Complexity of a Wrapper Update Query.* The main factor affecting the complexity of a wrapper update query is that its translation can lead to a set of database updates. To illustrate this, consider the update of a customer account according to the wrapper schema of Figure 2. Such an operation involves not only updating the `Account` column value in the table `Customer`, but also updating all the duplicated `Account` instances in the table `Order`.

When a set of database updates is needed, transaction techniques must be used to keep the legacy data consistency. The challenge is to permit wrapper updates to the underlying databases without violating their autonomy. Although this subject is somewhat beyond the scope of this article, we discuss it briefly for the sake of completeness. Transaction management can be viewed in two dimensions: autonomy and heterogeneity.

—*Autonomy.* It requires that the transaction management function of a wrapper be performed independently of the database transaction management execution functions. In other words, the database schema and rules are not modified to accommodate wrapper updates.
—*Heterogeneity.* It has the additional implication that the wrapper transaction manager of each database family may employ different concurrency control and commit protocols. Heterogeneity adds further difficulty since it becomes difficult to make uniform assumptions about the functionality provided by the legacy database. Some old database systems do not support any commit protocol. However, if a legacy database has ad hoc techniques that enable concurrent and recoverable access to a local data source, the wrapper can use them with minimal effort. Most recent database commit protocols contain specific operators, such as `begin`, `commit` and `abort`, which allow programmers to mark the code that is implied in a transaction. Other DBMS,

defined for advanced commit protocols, include more sophisticated behavior. For instance, in order to use the two-phase commit protocol [Gray 1993], a `prepare_to_commit` operator must be available.

3.5 Implicit Constraint Control

While the DBMS manages the explicit constraints defined in the physical schema, the wrapper emulates the implicit constraints by rejecting updates that violate them. To prevent inconsistencies, pretest services are implemented in the wrapper. For simplicity, the method is restricted to insertion or deletion of single instances of an entity type.

This method is based on the production, at wrapper development time, of implicit constraint checking components that are used subsequently to prevent the introduction of inconsistent data in the database. An implicit constraint checking is defined by a triple <ET,T,C> in which:

1. ET is an entity type of the physical schema;
2. T is an update type (e.g., insert or delete);
3. and C is an implicit constraint assertion ranging over entity type ET in an update of type T.

When an implicit constraint I is defined, a set of implicit constraint checking assertions can be produced for entity types used by I. Whenever an entity of type 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 associated with transformations that make implicit integrity constraints explicit during the reverse engineering phase that produces the wrapper schema (Section 5.2). To illustrate this concept, we consider the example of Figure 2. The implicit primary key of Order is associated with the triple <Order, INSERT, C> where C, defined in an SQL-like expression, is

```
"NOT EXISTS(SELECT * FROM Order WHERE Number = :Number)"
```

Assertion Enforcement Efficiency. Checking consistency assertions has a cost that depends on the physical constructs implemented in the legacy database. For instance, checking uniqueness (primary or candidate key) or inclusion (foreign key) assertions can be very costly if such a construct is not supported by an index. While legacy applications easily cope with missing indexes, for instance through sort/merge batch operations, new, generally transactional, applications cannot afford relying on such off-line batch procedures.

The order in which the assertions are evaluated is often relevant. The critical parameter to be considered is the evaluation cost. That is, the order depends among others, on the classes of the checking assertions and the amount of data access they involve. If no statistics information is available, we can only use heuristics 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) to the top of the list so that they are evaluated as early as possible.

Another approach consists in complementing the legacy database with new data structures, such as secondary indexes, intended to improve the performance of wrapper operations, and in particular assertion checking. Such techniques are not addressed in this article.

4. ABSTRACT SPECIFICATION OF A WRAPPER

Wrapping is a process that relies on schemas expressed according to different paradigms. Our previous work [Hainaut 2005] defined a wide spectrum entity-relationship model, the so-called *Generic Entity-Relationship Model* (GER) that can express data structure schemas whatever their underlying data model and their abstraction level. For instance, the GER is used to describe physical structures such as relational or COBOL file data structures, as well as canonical data structures (Figure 2).

An essential aspect of the GER is its ability to generate *submodels*, such as legacy physical and logical models through a three-step specialization mechanism. The generation of a model M from the GER proceeds in three steps. First, the constructs of the GER that are parts of M are retained, while the others are discarded. Second, the constructs are renamed according to the terminology of M. Third the construct assemblies that are valid in M are described by structural predicates. The core of the SQL2 relational model can be defined as follows.

1. *Subsetting.* SQL2 includes the following GER constructs: *entity type*, *attribute*, *identifier* and *inclusion constraint*. ISA relations and relationship types are ignored.
2. *Renaming.* An entity type is called a *table*, an attribute a *column*, an identifier a *candidate*/*primary key*, and the columns forming the LHS of an inclusion constraint a *foreign key*.
3. *Constraining*. An entity type comprises from 1 to (say) 254 attributes; attributes are atomic; attributes are single-valued (cardinality [0-1] or [1-1]).

Any GER schema that obeys these rules can be called SQL2-compliant. CODASYL, IMS, standard file, and XML Schema models, among others, have been defined similarly.

This kind of specialization brings an important advantage, that is, all intermodel transformations and conversions appear to be intramodel processes. As a consequence, a limited set of primitive operators is sufficient to model most database engineering processes, such as logical design, or reverse engineering. In wrapper generation, this also means that different legacy models can be treated with similar transformations.[4]

The way submodels are defined in the GER is similar to the concept of UML *profile*, though it appears to be more powerful and more expressive.

The GER is a high level model that encompasses in an abstract way, the constructs of a large family of models. Another approach, illustrated by McBrien

[4]For instance, strong similarities appear between the IMS hierarchical model of the 70s and XML Schema data structures. Consequently, many conceptual-to-logical transformations for IMS still are valid for producing XML structures.

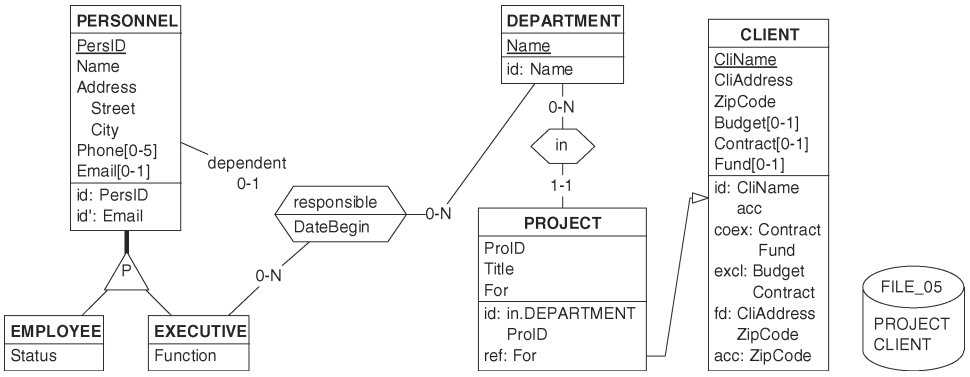


Fig. 5. Graphical representation of some GER constructs.

and Poulovassilis [1998], is based on low level models that comprise the intersection of such a family of models. Both approaches are discussed and compared in Hainaut [2005]. In this section, we present the main ideas of our previous work and extend it to show how the implicit structures and constraints can be specified by schema transformations on the GER.

4.1 Model and Schema Specification

For the needs of this article, the GER can be perceived as an enriched variant of the standard entity-relationship model that encompasses current physical, logical, and conceptual models. It is defined as a set of *constructs* comprising *structures* (including entity type, attribute, value domain, and relationship type) and *constraints*.

The major constructs of the GER are the entity types, the attributes, and the relationship types (Figure 5). Entity types are organized in single or multiple inheritance hierarchies. Attributes can be atomic (e.g., `Name`) or compound (e.g., `Address`), single-valued (e.g., `Address` or `Email`) or multi-valued (`Phone`). A cardinality constraint (e.g., `Phone[0-5]`) defines the minimum and maximum numbers of values associated with each parent instance, with default value `[1-1]`. Each role of a relationship type (e.g., `responsible`) can be labeled (e.g., `dependent`). It has a cardinality constraint stating the range of the number of relationships in which any entity can appear.

The GER includes a set of built-in constraints such as *primary identifiers* (e.g., `id: PersID`), *secondary identifiers* (e.g.,`id′: Email`), *foreign keys* (e.g., `PROJECT.(For)` → `CLIENT.(CliName)`, noted `ref: For`), generalized *inclusion constraints* and *functional dependencies* (e.g., `CLIENT: CliAddress` → `ZipCode`, noted `fd:CliAddress, ZipCode`). It also offers several existence constraints such as *coexistence* (all the components are null or all are not null; e.g., `coex(Contract, Fund)`), *exclusion* (at most one of the components; e.g., `excl (Budget,Contract)`), *at-least-one* (at least one of the components) and *exactly-one* (exactly one component). A semantic annotation can be associated with any object, for instance for specifying a complex constraint. The GER has no specific constraint language such as OCL. Instead, its N1NF relational semantics

allows complex constraints to be built through algebraic expressions [Hainaut 2005].

Constructs such as *access keys* (e.g., `acc: ZipCode`), which are abstractions of such structures as indexes and access paths, and *storage spaces* (e.g., `FILE_05`), which are abstractions of files and any other kinds of record repositories, are components of the GER as well.

Any concrete model, be it conceptual or physical, can be defined as a specialization of the GER.

4.2 Transformational Mapping Specification

The transformational approach has long been proposed as a sound basis for software engineering in general [Balzer 1981; Fikas 1985],[5] and for database engineering and in database design in particular [Rosenthal and Reine 1998]. Other processes, such as conceptual normalization [Rauh and Stickel 1995], optimization [Proper and Halpin 1998] and reverse engineering [Hainaut et al. 1993] have been modeled by transformation plans, notably to ensure semantics preservation. Transformations can be used to transform a schema from one model to another one, requiring specific operators for this couple of models.

In Hainaut [2005], we define a set of transformations valid for GER schemas. These transformations can be applied by a developer to build mappings between schemas expressed in the same or different data models. The use of the GER as the unifying data model allows constructs from different data modeling languages to be mixed in the same intermediate schema (as in the logical schema of Figure 2).

A transformation consists in deriving a target schema S' from a source schema S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty).

More formally, considering instance c of C and instance c' of C', a transformation Σ can be completely defined by a pair of mappings <T,t> such that $C' = T(C)$ and $c' = t(c)$. T is the structural mapping, which explains how to replace construct C with construct C' while t, the instance mapping, states how to compute instance c' of C' from any instance c of C.

A transformation can be specified through its *signature*, which 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.

4.2.1 *Inverse Transformation.* Each transformation $\Sigma_1 \equiv <T_1, t_1>$ can be given an inverse transformation $\Sigma_2 \equiv <T_2, t_2>$, usually denoted Σ^{-1}, such that, for any structure C, $T_2(T_1(C)) = C$.

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

[5]They consider that "*the process of developing a program [can be] formalized as a set of correctness-preserving transformations [. . .] aimed to compilable and efficient program production.*" Replacing the term *program* with *database schema* leads to a perfectly correct assertion.

only for a special variety of transformations,[6] called symmetrically reversible transformations. Σ_1 is said to be a symmetrically reversible transformation, or more simply semantics-preserving, if it is reversible and if its inverse is reversible too.

From now on, unless mentioned otherwise, we will work on the structural part of transformations, so that we will denote a transformation through its `T` part.

4.2.2 *Transformation Sequence.* A transformation sequence is a list of `n` primitive transformations: `S1-to-S2 = (T1 T2...Tn)`. For instance, the application of `S1-to-S2 = (T1 T2)` on a schema `S1` is defined by `S2 = T2(T1(S1))`.

As for schema transformation, a transformation sequence can be inverted. The inverse sequence `S2-to-S1` can be derived from the sequence `S1-to-S2` and can be defined as follows: if `S1-to-S2 = (T1 T2...Tn)` then `S2-to-S1 =` $(\mathtt{Tn}^{-1}\ldots\mathtt{T2}^{-1}\mathtt{T1}^{-1})$ *where* $\mathtt{Ti}^{-1}$ is the inverse of `Ti`; and hence `S1 = S2-to-S1(S2)`. In other words, `S2-to-S1` is obtained by replacing each origin schema transformation by its inverse and by reversing the operation order.

The concepts of sequence and its inverse are used for defining the direct and inverse mappings between two schemas. The transformational approach then consists in defining a (reversible) transformation sequence which, applied to the source schema, produces the target schema. The underlying sequence of structural mappings reflects the structural mappings between the two schemas whereas the chain of instance mappings reflects the correspondence at the data level.

The concept of transformation sequences, their properties and the processes that can be applied on them, such as inversion, slicing and agregation, are studied in Hainaut et al. [1996]. In particular, the condition under which two transformations can be swapped in a sequence without altering the effect of this sequence is precisely stated.

4.2.3 *Transformation Categories.* The notion of semantics of a schema has no generally agreed upon definition. We assume that the semantics of `S1` include the semantics of `S2` if and only if the application domain described by `S2` is a part of the domain represented by `S1`. Though intuitive and informal, this definition is sufficient for this presentation. In this context, three transformation categories can be distinguished:

— `T+` collects the transformations that augment the semantics of the schema (for example adding a constraint).
— `T=` is the category of transformations that preserve the semantics of the schema (for example transforming a foreign key into a relationship type).
— `T-` is the category of transformations that reduce the semantics of the schema (for example, discarding an attribute). These transformations allow defining a wrapper schema as a subset (view) of the physical schema. As such, the

[6]In Hainaut [2005], a proof system has been developed to evaluate the reversibility of a transformation.

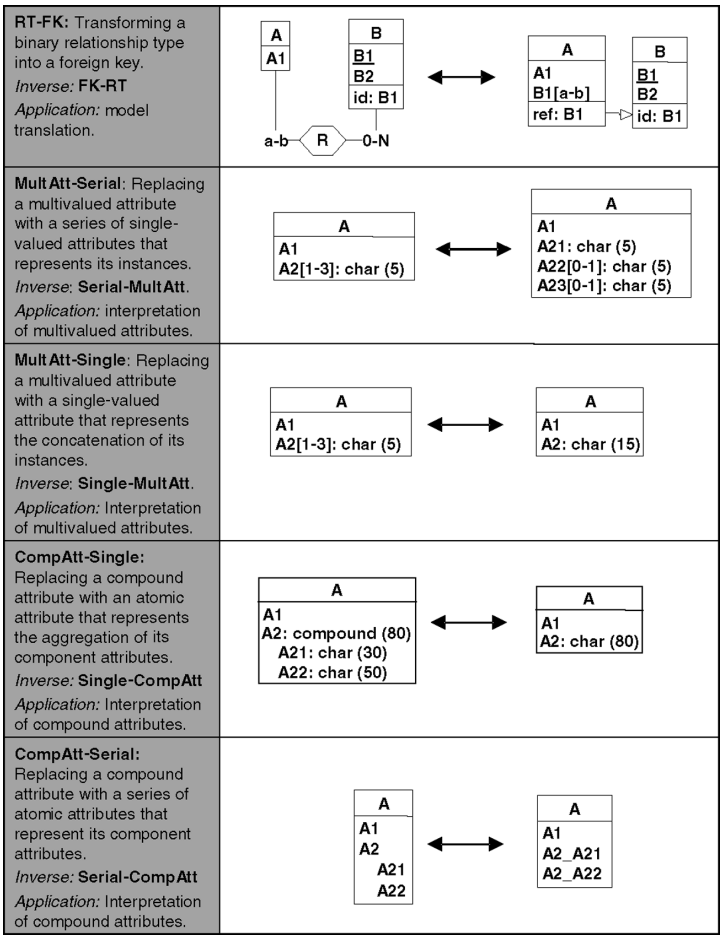


Fig. 6. Five transformation operators dealing with the interpretation of three structures.

usual problems associated with view updates must be addressed [Masunaga 1984].

To simplify the discussion, we assume that T+ applies on constraints only, whereas T= can transform structures and constraints. Moreover, we ignore the transformations of type T− because they are not relevant for our discussion. From now on, we will only consider transformations of types T+ and T=.

4.2.4 *Implicit Constraints and Schema Interpretation.* Transformations will be used for two different goals. First, T+ transformations will be used to make implicit constraints explicit. Second, producing the wrapper schema from the legacy logical schema involves applying T= transformations to improve the view the wrapper provides to the client applications, a process we will call *interpretation*.

We propose in Figure 6 and Figure 7, two sets of representative transformational operators. The first set is made up of T= transformations used for schema

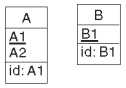
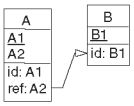
Schema Transformation	Database Schema	Wrapper Schema	Constraint Assertion
Create-Reference: A reference constraint is added. The referencing group and the group it references are made up of existing attributes.			**<A, INSERT, C1>** *Where* C1 ≡ EXISTS(SELECT * FROM B WHERE B.B1 = :A.A2) **<B, DELETE, C2>** *Where* C2 ≡ NOT EXISTS(SELECT * FROM A WHERE A.A2 = :B.B1)
Create-Identifier: An identifier group is added. The group is made up of existing attributes.			**<A, INSERT, C1>** *Where* C1 ≡ NOT EXISTS(SELECT * FROM A WHERE A.A1 = :A.A1)

Fig. 7. Transformation operators dealing with two typical implicit constraints.

interpretation, namely, transformation of a foreign key into a relationship type, and interpretation of multivalued attributes and compound attributes. These transformations are used in the query mapping [Thiran et al. 2005a]. The second set comprises transformations dealing with implicit constraints expression. Due to space limitations, the figure presents only two representative transformations, namely uniqueness and referential constraints, and sample associated constraint assertions.

These transformations will be used to build the wrapper schema from the physical schema, during the reverse engineering processes.

5. WRAPPER DEVELOPMENT

5.1 Generic Methodology

As we have seen through the previous section, wrapper specification addresses the challenge of legacy data heterogeneity and consistency through a generic data model, the GER, and a transformation paradigm that allow a description of query translation rules and implicit constraint checking at a high level of abstraction, independent of a specific source technology. In this section, we will describe the successive steps of a general methodology and of the tool components that support them. The key features of the approach can be summarized as follows:

—*Database reverse engineering.* Since most legacy databases have no up-to-date associated documentation, the latter must first be rebuilt through database reverse engineering techniques. These techniques yield all the necessary information to specify and develop the wrapper.

—*Semi-hardcoded wrapper.* A wrapper is developed as a program component dedicated to a specific database model and to a specific database. It comprises two parts, namely a model layer, in which the aspects specific to a given data model (e.g., RDB or standard files) are coped with, and a database layer that is dedicated to the specific database schema. While the model layer is common to all the databases built in this model, the wrapper/database schemas mapping is hardcoded rather than interpreted from

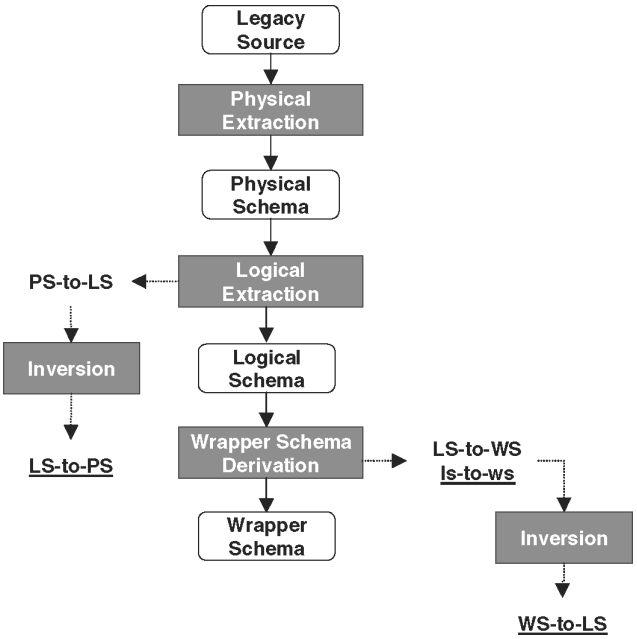


Fig. 8. The DBRE process yields schemas and mappings. The three mappings needed for developing the wrapper are underlined.

tables mapping as is the case in other approaches. Though such a wrapper may be larger than table-driven ones for large schemas, it provides better performance.

—*Schema transformation-based wrapper generation.* In Thiran et al. [2005a], the mapping rules were defined as schema transformations of type `T=`, which are used to automatically generate the internal query mapping and the data mapping. In this article, we use the same transformational paradigm, but extended to `T+` transformations, for specifying the implicit constraint assertions that have to be controlled by the wrapper.

—*Operational CASE support.* Schema processing, mapping processing and wrapper generation are each supported by a specific module of the CASE tool DB-MAIN.

5.2 Database Reverse Engineering for Schema Definition (DBRE)

In Section 2, we mentioned that the database schema, merely derived from the DDL code, most often is incomplete, and must be enriched with hidden constructs and constraints made explicit. To this end, we build on a proven approach, namely the DB-MAIN DBRE methodology. The key feature of this approach is threefold. First, all the schemas, whatever their DBMS and their abstraction level, are expressed in the GER. Second, it uses the same transformational approach than that of this article. Third, this approach is supported by an operational CASE tool.

The execution of the methodology produces two result types (Figure 8): (1) the wrapper schema, including implicit and explicit constructs expressed in the

canonical data model; and (2) the schema transformation sequences (and their inverse) applied on the physical schema to get the wrapper schema.

Since this methodology has been presented in former papers (Hainaut [2002] and Hainaut et al. [1993]), we will only recall those processes that are relevant to wrapper development.

5.2.1 *Physical Extraction.* This phase consists in recovering the (existing) physical schema made up of all the structures and constraints explicitly declared. This process is often easy to automate since it can be carried out by a simple parser that analyses the DDL texts, extracts the data structures, and expresses them as the physical schema.

5.2.2 *Logical Extraction.* This process relies on four main sources that can bring information of various certainty levels. Some of them can be used to find clues of possible implicit constructs while others can confirm or discard such hypotheses. We will briefly discuss these sources and some of the associated techniques.

The simplest, but not so reliable, techniques consist in analyzing the physical schema. For instance, name structure and data type similarities can suggest a unique or foreign key. The nonkey attribute `Customer` of the table `Order`, the type of which is the same as that of the primary key `Reference` of the table `Customer` may be a foreign key to the latter. Frequent physical design heuristics can also be used. For example, most file and database developers associate an index with each unique key and with most foreign keys. Therefore, the attribute `Customer` is more liable to be a foreign key if it is supported by an index.

Data analysis can bring stronger information, but is best performed when some evidence has already been obtained through other techniques. It consists in analyzing sample data to check whether or not a definite property holds. Such techniques are very efficient for finding functional dependencies, uniqueness and referential constraints and to discover the value set of enumerated domains [Novelli and Cicchetti 2001].

Program code analysis certainly is the most complex approach, but it brings the most valuable information as well [Yang and Bennett 1995]. The idea is simple: the way external data are processed in the application programs strongly depends on the properties of these data. Therefore, understanding the logic that underlies the processing of external data leads to the understanding of many implicit data structures and constraints. The examination of the code of transactions [Ritsch and Sneed 1993], or of the shape of SQL queries [Petit et al. 1994; Lopes et al. 2002], generally brings important knowledge on field value constraints, on correlation among values of different fields, or on interfile properties such as referential integrity. Dependency graph analysis can show hidden relationships between data fields [Henrard and Hainaut 2001]. The mere observation of assignment statements can bring information on implicitly structured fields. One of the most common examples is that of anonymous fields (filler in COBOL or concatenated values in a relational column). Let us suppose that an application component includes a statement such as:

```
MOVE ADD OF CUSTOMER-RECORD TO TMP-ADDR
```

where CUSTOMER-RECORD is the name of an internal compound variable associated with table CUSTOMER, and in which the subfield ADD received the value of atomic column Address, and TMP-ADDR that of another internal compound variable made up of subfields (STREET, ZIP-CODE, CITY). This statement suggests that the same substructure also applies to the source column Address.

More powerful program understanding techniques, such as *program slicing* are used to elicit complex relationships between fields. From the original proposal [Weiser 1984], specific techniques have been developed for data structure understanding. For instance, Henrard and Hainaut [2001] show how this technique can be used to collect the statements that contribute to the state of a record at a given point in the program. This technique is more precise than mere transaction analysis, since it gives the analyst a clear view of the code fragment from which data properties can be inferred, though the statements are scattered throughout thousands of lines-of-code (LOC).

Finally, the analysis of input/output documents and forms generally gives precise information on format, constraints, relationships and interpretation of data fields. Indeed, many reports and electronic forms, for data entry for example, are some kind of views on the data [Lee and Yoo 2000].

5.2.3 *Wrapper Schema Derivation.* This process of semantic interpretation consists in exporting and interpreting the logical schema, from which one tries to extract the wrapper schema WS and the schema transformation sequence LS-to-WS. Two main different problems have to be solved through specific techniques and reasoning:

—*Model translation*: the logical schema expressed in the GER must be expressed in the operational data model of the wrapper. This process can be fairly straightforward if the logical and wrapper models are similar (e.g., DB2-to-ODBC), but it can be quite complex if they are different (e.g., Oracle-to-XML or COBOL-to-relational). Model translation basically is a schema transformation. It consists in translating a schema expressed in a source data model Ms into a schema expressed in a target data model Mt, where Ms and Mt are defined as two submodels of the GER. Model transformation is defined as a model-driven transformation within the GER. A model-driven transformation consists in applying the relevant transformations on the relevant constructs of the schema expressed in Ms in such a way that the final result complies with Mt [Hainaut 2005]. Operators RT-FK and CompAtt-Serial (Figure 6) are typical conceptual-to-relational transformations. Therefore, their inverses are often used in wrapper schema derivation.

—*Deoptimization*: most developers introduce, intentionally or not, optimization constructs and transformations in their physical schemas. These practices can be classified in three families, namely structural redundancies (adding derivable constructs, such as column Account in table Order in Figure 2), unnormalization (merging data units linked through many-to-one relations), and restructuring (such as splitting and merging tables). The deoptimization process consists in identifying such patterns, and discarding them, either through removal or by transformation. Operators MultAtt-Serial and

MultAtt-Single (Figure 6) are examples of relational optimizations. Consequently, their inverses Serial-MultAtt and Single-MultAtt are deoptimization transformations.

5.2.4 *Mapping Definition.* The production of the wrapper schema (WS) from the physical schema (PS) defined in two distinct or identical models, can be described by the sequence of transformations PS-to-WS in such a way that: WS = PS-to-WS(PS) where PS-to-WS = (PS-to-LS LS-to-WS). The inverse of transformation PS-to-LS is LS-to-PS, while that of LS-to-WS is WS-to-LS. Therefore, the inverse of PS-to-WS is WS-to-PS = (WS-to-LS LS-to-PS).

As an illustration of mapping definition, we consider the mapping definition between the physical and wrapper schemas of Figure 3. For readability, we only consider the most relevant transformations by means of their (simplified) signature:

```
T1: () ← Create-Id(Customer,(Reference))
T2: () ← Create-Id(Order,(Number))
T3: () ← Create-Ref(Order,(Customer),Customer,(Reference))
T4: (Address) ← SingleCompAtt(Customer,(Street, Zip, City))
```

The transformation sequence PS-to-WS is then defined as follows: PS-to-WS = (T1 T2 T3 T4). The wrapper schema (WS) of the Figure 3 is obtained by the application of the transformation sequence PS-to-WS on PS:WS = T4(T3(T2(T1(PS)))). The inverse sequence is defined as follows: WS-to-PS = ($T4^{-1}T3^{-1}T2^{-1}T1^{-1}$).

5.3 Wrapper Generation

At this stage, we are provided with the necessary specifications to produce the wrapper. First, the legacy physical, the legacy logical, and the wrapper schemas are completely specified. Their data structures will be the basis of the internal data structures of the wrapper and of the *query/update analyzer*. Second, the inverse of the physical-to-logical mappings (LS-to-PS) is formally defined; it will be used to code the *implicit constraints control* component. Third, the structural part of the wrapper-to-logical mapping (WS-to-LS) will be used to code the *query/update translator*, while the instance part of its inverse (ls-to-ws) will define the *data translator*.

The relations between the mapping specifications and the wrapper functions are depicted in Figure 9. Since the mappings are based on a limited but comprehensive set T of transformations that are proved to be correct, all the wrappers that are built according to this approach can be considered *correct* and *complete* with respect to T (provided the code generation algorithms based on T are carefully designed).

5.4 Tool Support

The generation of R/W wrappers is supported by the DB-MAIN tool. DB-MAIN is a graphical, general-purpose, programmable, CASE environment dedicated to *database application engineering*. Besides standard functions such as schema

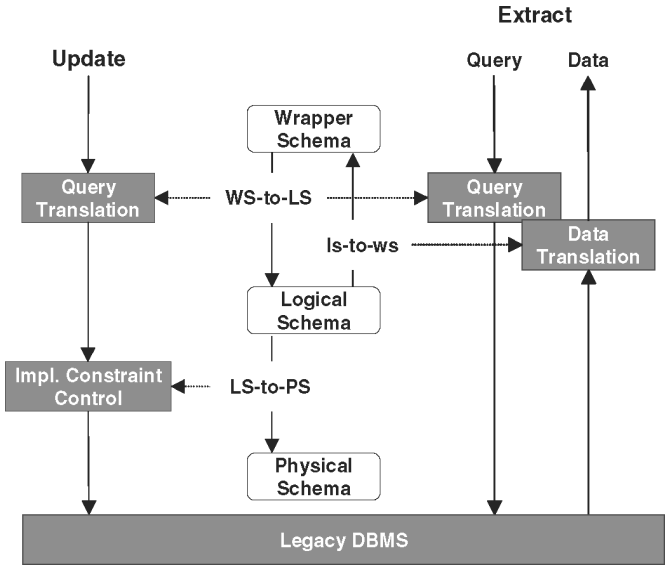


Fig. 9. Three mappings built during the reverse engineering process are used to develop major components of the wrapper.

entry, examination and management, it includes advanced processors such as DDL parsers, transformation toolboxes, reverse engineering processors and schema analysis tools. An interesting aspect of DB-MAIN is its meta-CASE layer through which new models, new methodologies, and new processors can be developed. The latter are written either in Java or in the specialized language Voyager 2, and appear as extensions (plugins) of the CASE tool. Figure 10 depicts the main components of the architecture of DB-MAIN. It shows the role of the Voyager 2 abstract machine, that executes the extensions requested by the command interpreter. The method engine controls the engineering processes defined by the method engineer, and the definitions of which are stored in the repository. Two Voyager 2 plugins are of particular interest for wrapper development, namely an SQL DDL code analyzer that supports physical extraction (Section 5.2.1) and the wrapper generator (Section 5.3).

Further details on DB-MAIN can be found in Englebert and Hainaut [1999] and Hick [2005]. In the limited scope of this article, we describe some of the DB-MAIN assistants dedicated to schema definition and wrapper code generation only.

5.4.1 *Schema Building. Extraction facilities.* Database schemas can be extracted by a series of processors which identify and parse the declaration part of the DDL source texts, or analyze catalog tables, and create corresponding abstractions in the repository. Extractors have been developed for SQL, COBOL, IDMS, IDS2, IMS/DL1, RPG, and XML DTD data structures. Additional extractors can be developed easily thanks to the *Voyager 2* environment.

Logical schema extraction and wrapper schema derivation. These processes heavily rely on transformation techniques. For some fine-grained reasonings,

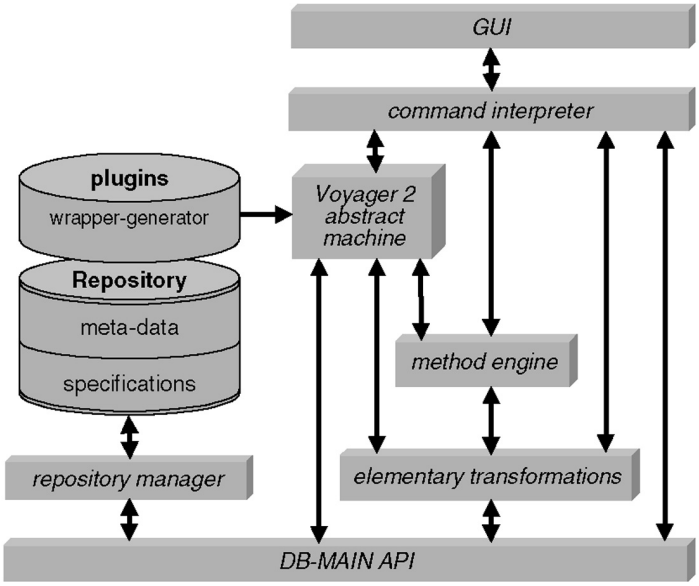


Fig. 10. Architecture of the DB-MAIN CASE environment.

precise surgical transformations have to be carried out on individual constructs. This is a typical way of working in refinement tasks. In the case of model translation, some heuristics can be identified and materialized into a transformation plan. DB-MAIN offers a dozen predefined model-based transformations including ER, UML Class diagrams, SQL, COBOL and XML translation from other models.

5.4.2 *Wrapper Code Generation. History analyzer.* DB-MAIN automatically generates and maintains a history log of all the transformations that are applied when the developer carries out any engineering process such as wrapper schema definition. This history is completely formalized in such a way that it can be analyzed, transformed and inverted. A history basically is a procedural description of inter-schema mappings. The history analyzer parses history logs and transforms them into nonprocedural annotations that define the interschema object mappings.

Wrapper encoders. The implicit constraints emulation code is derived from the constraint assertions that are associated with the transformations of Figure 7, while the model translation sections (query/update analyzer and query translation) are generated from the nonprocedural annotations. The data translation section is derived from the procedural conversion of the `ls-to-ws` mapping.

The wrappers are generated for two server protocols, namely SQL-based through a variant of JDBC, and object-based. At the current time, *Voyager 2* wrapper encoders for COBOL files and relational data structures are available.

Table I. The Two Case Studies And Their Size

Application	Wrapper Schema Size	Transformation Sequence Size	Implicit Constructs
A	3 entity types 15 attributes	3 transformations	3 implicit constraints
B	30 entity types 120 attributes	30 transformations	30 implicit constraints

6. EXPERIMENTS

The approach described in this article has been applied on several actual systems; two of them are briefly described in this section. The first application (A-COB) is a small size COBOL test bed we have developed to precisely check the various versions of our generator. It includes a 3-file database that comprises examples of complex hidden structures and constraints, together with a 400 LOC application program. The second application (B-RDB) is a collection of similar INFORMIX relational databases dedicated to tax management in a Belgian municipality and exhibiting complex redundancy patterns. Wrappers of application B-RDB have been integrated into a database federation controlled through a light mediator developed in JAVA/HTML. The latter provided some functions to arbitrate among conflicting data from the tax databases.

We have also migrated each of them in the other technology, which was a straightforward process, since both databases comprise flat files/tables only. This provides us with two additional case studies, namely A-RDB and B-COB.

According to the architecture described in Thiran and Hainaut [2001] and recalled in this article, the size of a wrapper is the sum of the LOC of the model layer and of that of the database layer. The first layer has a constant size, which is, for the current version of the generators, of 7,500 LOC for RDB wrappers and 4,400 LOC for COBOL wrappers. Evaluating the cost of the database layer is more complex, since it depends of several factors:

—the underlying DBMS to which the wrapper is dedicated;
—the size of the database and wrapper schemas;
—the number and the type of schema transformations of the sequence.

Table I specifies, for applications A and B, the composition of the wrapper schema (number of entity types and attributes), the number of transformations that define the mappings and the number of implicit constraints that have been emulated. Note that, due to the simplicity of the database schemas (flat structures only), these figures are valid for both COBOL and SQL technologies, and therefore apply to the four case studies.

Table II gives the size of the code fragment (COBOL for application A, and C for application B) that is generated for each of the following constructs of the wrapper schema: entity type, attribute, implicit identifier, and implicit foreign key. The table should be augmented with the score of additional constructs for other case studies.

The size of the wrappers can be computed from these tables. For instance, the size of the wrapper of application B is about 16,000 LOC in its SQL2/C variant

Table II. LOC Size of Explicit Constructs and Implicit Constraints

Constructs	COBOL Wrapper	RDBMS Wrapper
Entity type	380	125
Attribute	120	40
Implicit identifier	20	30
Implicit reference	15	15

(B-RDB), while it is 17,000 LOC in its file/COBOL variant. The differences between both technologies stem from the way the database layers are generated. In COBOL wrappers, each implicit constraint is managed by a specific code fragment in COBOL while it is emulated by an SQL query in RDB wrappers (in some sense, the emulation is hard-coded in the wrapper).

To date, we have not carried out any intensive performance measurements on the use of wrappers. The feeling we have gotten from testing various applications on both case studies leads us to the following conclusions. First, we have observed some performance degradation; but that never exceeded 15% for intensive data access applications, and that obviously came from the interpretation layers (i.e., the CPU resource). Second, the degradation is largely dependent on the logic of the client application, so that drawing general conclusions without modeling application profiles seems useless. Third, the structure of the wrapper shows that the performance is not significantly dependent on the size of the schema.

7. CONCLUSIONS AND PERSPECTIVES

Wrappers that make legacy and deficient databases comply with the requirements of new applications have proved to be the core technology that allows a smooth transition to modern architectures. One the one hand, by converting a legacy model, such as IMS, CODASYL or early relational structures, to relational, object, or XML structures better fitted to the interoperable architectures, a wrapper makes the integration of a legacy database into current large applications easier. On the other hand, by emulating the implicit constraints and structures that are not managed by the legacy DBMS, they relieve modern client components from the responsibility of controlling data integrity. The system comprising the legacy database, and the wrapper, form a dedicated but modern DBMS. The newly developed client components can update the legacy data without worrying about data integrity, as is standard with current database servers.

It appears that a database wrapper is not a simple component. Indeed, it is based on two, possibly quite different, data models, and is in charge of efficiently translating queries, updates and data between both. Moreover, it has to synchronize different APIs, that possibly have quite different approaches to managing currency states in sequenced data access. Matching relational cursors, ODBC handles with CODASYL currency registers, or COBOL implicit current record, implies complex dynamic correspondence that often requires a third current management system (the internal wrapper currents). Error recovery and transaction management can also be wrapper responsibilities. Finally,

a wrapper can be a large piece of code that cannot be handwritten, but for small scale systems.

Hence the importance of precisely defining the architecture of such R/W wrappers and of elaborating methodologies for their rapid and reliable development. However, the current state of the art does not provide us with many applicable proposals for solving these architectural and methodological problems.

The approach that has been presented in this article is an attempt to solve them in a generic framework that is model and technology independent. Based on reverse engineering techniques, it produces the major components of the R/W wrappers dedicated to a legacy database, namely the physical schema, the complete logical schema, which includes the implicit constructs, the wrapper schema, and the inter-schema mappings. Since these results are completely formalized, they can be used for automatically generating the code of the wrapper. This function is ensured by the DB-MAIN CASE environment.

However, complete automation of the production of R/W wrappers is clearly unrealistic. Indeed, though some of the most common implicit constraints and structures have been identified and formalized (Section 4.2), and therefore are candidates for full automation, the range of idiosyncrasies and non standard constructs that can be found in actual databases is so large [Blaha and Premerlani 1995], that providing room for hand-writing some small specific code sections in the body of a wrapper, is a necessity.

The methodology and its supporting tool have been evaluated in a few academic and industrial case studies based on COBOL files and (possibly early) relational databases. Through them, we have learned that the approach is both applicable and efficient, particularly in large scale databases. The part of the hand-writing wrapper code was easily identifiable and its complexity was acceptable.

Experiments have also been carried out in application areas of federated databases (for a city administration system) and data migration (from legacy databases to XML [Thiran et al. 2005b]).

Since 2003, we have been integrating this wrapper technology into a development environment for business-to-customer applications that are built on top of legacy databases. At the same time, we are investigating some important issues that have not been tackled so far, such as optimization processing and transaction management.

REFERENCES

AIKEN, P., 1996. *Data Reverse Engineering*. McGraw-Hill.

BALZER, R. 1981. Transformational implementation : An example. *IEEE TSE 7*, 1, 3–14.

BERGAMASCHI, S., CASTANO, S., BENEVENTANO, D., AND VINCI, M. 2001. Semantic Integration of Heterogeneous Information Sources. *Data Knowl. Eng. 36,* Elsevier, 215–249.

BLAHA, M. R. AND PREMERLANI, W. J. 1995. Observed Idiosyncrasies of Relational Database designs. In *Proceedings of the 2nd IEEE Working Conference on Reverse Engineering*, Toronto, July, IEEE Computer Society Press.

BOUGUETTAYA, A., BENETALLAH, B., AND ELMAGARMID, A. 1998. *Interconnecting Heterogeneous Information Systems.* Kluwer Academic Publishers.

BRODIE, M. AND STONEBRAKER, M. 1995. *Migrating Legacy Systems*, Morgan Kaufmann.

CAREY, M. J., FLORESCU, D., ZACHARY, G. I., AND YING, L. 2000. XPERANTO: Publishing Object-Relational Data as XML. In *Proceedings of WebDB* (Informal Proceedings), 105–110.

EDWARDS, H. AND MUNRO, M. 1995. Deriving a Logical Model for a System Using Recast Method. In *Proceedings of the 2nd WCRE Conference*, Toronto, IEEE Computer Society Press.

ENGLEBERT, V. AND HAINAUT, J.-L. 1999. DB-MAIN: A Next Generation Meta-CASE. *Inform. Syst. J. 24*, 2. Pergamon, 99–112.

FERNANDEZ, M., TAN, W., AND SUCIU, D. 2000. Silkroute: Trading between Relations and XML. *Comput. Netw. 33*, Elsevier, 723–745.

FIKAS, S., F. 1985. Automating the Transformational Development of Software, *IEEE TSE 11*, 1268–1277.

GARCIA-MOLINA, H., PAPAKONSTANTINOU, Y., QUASS, D., RAJARAMAN, A., SAGIV, Y., ULLMAN, J. D., VASSALOS, V., AND WIDOM, J. 1997. The TSIMMIS Approach to Mediation: Data Models and Languages. *J. Intell. Inform. Syst. 8*, 2, 117–132.

GRAY, P. 1993. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv. 25*, 2, 73–170.

HAINAUT, J.-L., CHANDELON M., TONNEAU, C., AND JORIS, M. 1993. Transformational techniques for database reverse engineering. In *Proceedings of the 12th International Conference on ER Approach*, ER Institute, 364–375.

HAINAUT, J.-L., HENRARD, J., ROLAND, D., AND ENGLEBERT, V. 1996. Database Design Recovery, in *Proceedings of the 8th Conference on Advanced Information Systems Engineering (CAiSE)*, Springer-Verlag, 272–300.

HAINAUT, J.-L. 2002. Introduction to Database Reverse Engineering. *LIBD Lecture Notes*, University of Namur. {http://www.info.fundp.ac.be/∼dbm/publication/2002/DBRE-2002.pdf; October 2005}.

HAINAUT, J.-L. 2005. Transformation-Based Database Engineering. In *Transformation of Knowledge, Information and Data: Theory and Applications*. P. van Bommel, Ed. IDEA Group, 1–28.

HENRARD, J. AND HAINAUT, J.-L. 2001. Data Dependency Elicitation in Database Reverse Engineering. In *Proceedings of the 5th European Conference on Software Maintenance and Reengineering* (*CSMR*), IEEE Computer Society Press, 11–19.

HENRARD, J., HICK, J.-M. THIRAN, Ph., AND HAINAUT, J.-L. 2002. Strategies for Data Reengineering. In *Proceedings of the WCRE Conference*, Richmond, IEEE Computer Society Press, 211–220.

HICK, J.-M., ENGLEBERT, V., HENRARD, J., ROLAND, D., AND HAINAUT, J.-L. 2004. The DB-MAIN Database Engineering CASE Tool (version 7.1)—Functions Overview, *DB-MAIN Technical Manual*, Institut d'Informatique, University of Namur. {http://www.reversa.com/DISTRIBUTION/VERSION.7/DB-MAIN-Reference-Manual.pdf; October 2005}

LAWRENCE, R., BARKER, K., AND ADIL, A. 1998. Simulating MDBS Transaction Management Protocols. In *Proceedings of the CAINE Conference*.

LEE, H. AND YOO, C. 2000. A Form Driven Object-Oriented Reverse Engineering Methodology, *Inform. Syst. 25*, 3. Elsevier, 235–259.

LIM, E. P. AND LEE, H. K. 1999. Export Database Derivation in Object-Oriented Wrappers. *Inform. Softw. Tech. 41*, Elsevier, 183–196.

LOPES, S., PETIT, J.-M., AND TOUMANI, F. 2002. Discovering Interesting Inclusion Dependencies: application to logical database tuning, *Inform. Syst. 27*, Elsevier, 1–19.

MASUNAGA, Y. 1984. A Relational Database View Update Translation Mechanism. In *Proceedings of the 10th International Conference on Very Large Data Bases*, Morgan Kaufmann, 309–320.

MCBRIEN, P. AND POULOVASSILIS, A. 1998. A General Formal Framework for Schema Transformation, *Data Knowl. Eng. 28*, 1, Elsevier, 47–71.

NOVELLI, N. AND CICCHETTI, R. 2001. Functional and Embedded Dependency Inference: a Data Mining Point of View, *Inform. Syst. 26*, 477–506.

PETIT, J.-M., KOULOUMDJAN, J., BOULIAUT, J.-F., AND TOUMANI, F. 1994. Using Queries for Improving Database Reverse Engineering. In *Proceedings of the 13th International Conference on ER Approach*, Manchester, Springer-Verlag.

PROPER, H. A. AND HALPIN, T. A. 1998. Database Schema Transformation & Optimization. In *Proceedings of the 14th International Conference on Conceptual Modeling*, LNCS, 1021, Springer, 191–203.

RAUH, O. AND STICKEL, E. 1995. Standard Transformations for the Normalization of ER Schemata. In *Proceedings of the CAiSE95 Conference*, Jyväskylä, Finland, LNCS, Springer-Verlag, 313–326.

RITSCH, H. AND SNEED, H. 1993. Reverse Engineering Programs via Dynamic Analysis. In *Proceedings of the 1st IEEE Working Conf. on Reverse Engineering*. Baltimore, IEEE Computer Society Press, 192–201.

ROSENTHAL, A. AND REINER, D. 1988. Theoretical Sound Transformations for Practical Database Design. In *Proceedings of Entity-Relationship Approach*. LNCS, Springer-Verlag, 115–131.

ROSS, W. 1993. Hewlett-Packard's Migration to Client/Server Architecture. In *Distributed Computing: Implementation and Management Strategies,* Prentice Hall, ed. Khanna, M.

ROTH, M. AND SCHWARZ, P. 1997. Don't Scrap It, Wrap it! A Wrapper Architecture for Legacy Data Sources. In *Proceedings of the VLDB Conference.* Morgan Kaufmann, 266–275.

SHANMUGASUNDARAM, J., KIERNAN, J., SHEKITA, E. J., FAN, C., AND FUNDERBURK, J. 2001. Querying XML Views of Relational Data. In *Proceedings of the 27th VLDB Conference*. Morgan Kaufmann, 261–270.

SOUDER, T. AND MANCORIDIS, S. 1999. A Tool for Securely Integrating Legacy Systems into Distributed Environment. In *Proceedings of the 6th WCRE Conference*. IEEE CS Press, 47–55.

THIRAN, Ph. AND HAINAUT, J.-L. 2001. Wrapper Development for Legacy Data Reuse. In *Proceedings of the WCRE Conference*. Stuttgart, Germany, October, IEEE CS Press, 198–207.

THIRAN, Ph., HOUBEN, G.-J. , HAINAUT J.-L. AND BENSLIMANE, D. 2004. Updating Legacy Databases through Wrappers: Data Consistency Management. In *Proceedings of WCRE'04*, IEEE CS Press, 58–67.

THIRAN, Ph., HAINAUT, J.-L., HOUBEN, G.-J. 2005a. Database Wrappers Development. Towards Automatic Generation. In *Proceedings of the CSMR Conference*. Manchester, March, IEEE Computer Society Press, 207–216.

THIRAN, Ph., ESTIEVENART, F., HAINAUT, J.-L. AND HOUBEN, G. J. 2005b. A Generic Framework for Extracting XML Data from Legacy Databases. *Journal of Web Engineering, 4*, 3, Rinton Press, 205–223.

WEISER, M. 1984. Program Slicing, *IEEE TSE 10*, 352–357.

WU, B., LAWLESS, D., BISBAL, J., GRIMSON, J., WAD, V., O'SULLIVAN, D., AND RICHARDSON, R. 1997. Legacy System Migration: A Legacy Data Migration Engine, In *Proceedings of the 17th International Database Conference (DATASEM '97)*, Ed. Czechoslovak Computer Experts, 129–138.

YANG, H. AND BENNETT, K. H. 1995. Acquisition of ERA Models from Data Intensive Code. In *Proceedings of the International Conference on Software Maintenance, ICSM '95*, Opio (Nice), France, October 17–20. IEEE Computer Society, 116–123.

Received April 2005; revised November 2005, December 2005; accepted January 2006