Wrapper Development for Legacy Data Reuse

Ph. Thiran and J.-L. Hainaut
InterDB Project, Database Applications Engineering Laboratory
Institut d'Informatique, University of Namur
rue Grandgagnage, 21 - B-5000 Namur - Belgium
{pth,jlh}@info.fundp.ac.be

Abstract

This paper is devoted to the technology of wrappers for legacy data systems reuse. Their characteristics are outlined and a generic wrapper architecture is defined. This architecture is intended to be instanciated for specific legacy data models and systems. A general methodology is proposed to define the architecture components. The methodology is supported by an operational CASE-tool that helps developers to generate wrappers.

1. Introduction

Roughly speaking, a wrapper is a software component that converts data and queries from one data model, generally the DMS[1] model, to another, DMS-independent, model ([2], [12], [18], [17]). That is, the wrapper (1) offers an export schema of an existing database, expressed in an abstract data model (often called canonical), (2) accepts queries against the export schema and translates them into queries understandable by the underlying DMS, and (3) transforms the result of the queries into a format that complies with the export schema.

Other definitions are based on the encapsulating of the procedural components (see [4], [14], [19]) as well. In this paper, we consider only the data encapsulation.

1.1. Wrapper and legacy data system

Wrapping legacy data systems poses complex problems ([3], [11]). Legacy data systems can be based on various data models, ranging from standard file structures to relational and object models. To deal with this heterogeneity, a wrapper must hide the model that a legacy system implements by providing a more abstract and common model, in other words, a canonical model. Such a model must be highly generic and more flexible than the legacy data models [5].

Data models of such systems cannot express all the semantics of the real world. Limitations of the modeling concepts and information hiding programming practices lead to the incompleteness of the physical schema [13]. Therefore, a wrapper cannot simply inherits the poor quality and the incompleteness of the physical schemas of the legacy data. The wrapper must often offer a semantically richer description of a 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. Hence, the close link between wrapper development and database reverse engineering (DBRE), one of the goals of which is to elicit hidden structures and constraints.

Finally, different languages are used to manipulate data represented in different data models. The query capabilities of these languages have multiple and various functionalities (e.g., COBOL I/O statements *vs* SQL queries). A wrapper translates commands, or queries, from one language to another one. This is not always possible to translate a wrapper query into a single legacy query. Due to the limited functionalities of some DMS (Data Management System), the wrapper must often simulate operations and behaviour required by the canonical model. Let us consider two examples.

- If the canonical model includes foreign keys or inter-object relationships, then a wrapper for COBOL files must simulate the associated delete and update modes, that tell how to propagate and control these operations.
- If the canonical model offers a SQL-like language, then a wrapper for COBOL files must simulate at least some primitive forms of selection predicate, for instance those that involve the fields of the record types, be they supported by an index or not.

1 DMS standing for Data Management System, a term encompassing both File systems and Database systems.

1.2. Paper scope and outline

The paper concentrates on defining a generic architecture for wrappers dedicated to legacy data systems. It proposes also a methodology intended to define the architecture components, including schema recovery and mapping building. The methodology is supported by the DB-Main CASE tool that helps to build architecture components in a systematic way.

The paper is organized as follows. Based on our experience in data wrapping, Section 2 presents the main baselines of a generic architecture of wrappers. Section 3 and 4 develop two important concepts of the architecture: the generic model and the mapping definition. In Section 5, we present the architecture components of wrappers and their characteristics. Section 6 proposes a methodology for building the architecture components. In section 7, we briefly present the role of CASE technology in order to support this methodology. Section 8 concludes the paper.

2. Wrapper baselines

2.1. InterDB project

As a part of the InterDB project[1], dedicated to Heterogeneous Database Interoperability [15], we built hard-coded wrappers for several legacy data systems, including relational databases and COBOL files. Wrappers that are more than 10,000 LOC long are not uncommon, so that developing them represents an important effort in extending, integrating and reusing legacy systems. Providing models, techniques and supporting tools for wrapper development quickly became an obvious goal of the project.

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 [15] and [16], we also demonstrated that the code specific to a particular legacy data system performs the structural and instance mappings and that these mappings can be modeled through semantics-preserving transformations [8]. 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 experiences in our current application project areas of city administration systems, we have also stated that the formalized mapping cannot cover all the complexity of a data source (for instance, conflicts occuring among the 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.

2.2. Wrapper dimensions and schemas

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

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 basic wrapper. The basic wrapper wraps the legacy data system and offers an interface based on the component schema of the wrapped data system. The basic wrapper converts data and queries from the legacy data model (LDM) to the wrapper data model (WDM). The basic wrapper relies on schemas descriptions and mappings to translate queries and to form the result instances. That is, they can be complex if the mapping rules are complex and the wrapper data model is rich. As a result, a realistic basic wrapper should be based on an operational model, such as SQL2 or OO, and a realistic set of mapping rules.

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

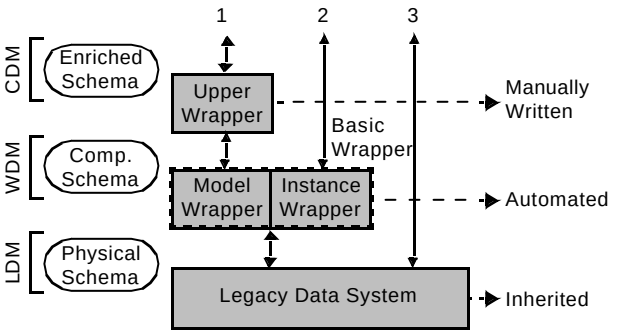

Figure 1. The three dimensions and schemas of a wrapper and the three ways for accessing a legacy data system.

The upper wrapper is built on top of these two components. It offers an enriched view of the component schema.

1 The InterDB project is supported by the Belgian *Région Wallonne.*

This view is based on a model (CDM) that is highly generic and more flexible than the wrapper and legacy data models. The complex mapping rules (that cannot be taken into account by the basic wrapper) are programmed manually

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

Example 1. Let us assume that the wrapper model cannot express an (implicit) constraint like: *"the delivery date must be subsequent to the order date"*. This constraint cannot be generated as a part of the instance wrapper and has to be coded in the upper wrapper

3. Wrapper schemas and generic model

Developing wrappers is a process that relies on a hierarchy of schemas at various levels of abstraction and acording to several paradigms. In the proposed approach, schema are expressed in a unique wide spectrum specification model, the so-called *generic model*, from which the legacy data models, the wrapper and the canonical model can be derived by specialization. In short, physical schemas, component schemas as well as enriched schemas are expressed into a unique and generic entity/object-relationship model we will describe below.

3.1. The generic model

The main concepts of the generic model are illustrated graphically in Figure 2. The central construct is that of entity type, or object type (CUSTOMER), that represents any homogeneous class of conceptual, component or physical entities, according to the abstraction level at which these entities are perceived. Entity types can have attributes (CustCode, Price, UnitPrice), which can be atomic (QtyOH) or compound (Price), single-valued (Name) or multivalued (Price), mandatory (Name) or optional (Phone). Cardinality [i-j] of an attribute specifies how many values (from i to j) of this attribute must be associated with each parent instance (entity or compound value). The values of some attributes, called reference attributes (DETAIL.ItemCode), can be used to denote other entities (i.e., they form some kind of foreign keys). Relationship types (places, has) can be drawn between entity types. Each of their roles (places.ORDER) is characterized by a cardinality constraint [i-j], stating that each entity must appear in i to j relationships. Additional constraints such as identifiers made of attributes and/or roles as well as existence constraints (coexistence, exclusive, at-least-one, etc.) can be defined. Constructs such as access keys (ITEM.{Name}), which are abstractions of such structures as indexes and access paths, and storage spaces (File_DOC) which are abstractions of files and any other kinds of record repositories, are components of the generic model as well. A processing unit (CUSTOMER.Remove) is the abstraction of a program, a procedure or a method, and can be attached to an entity type, a relationship type or a schema.

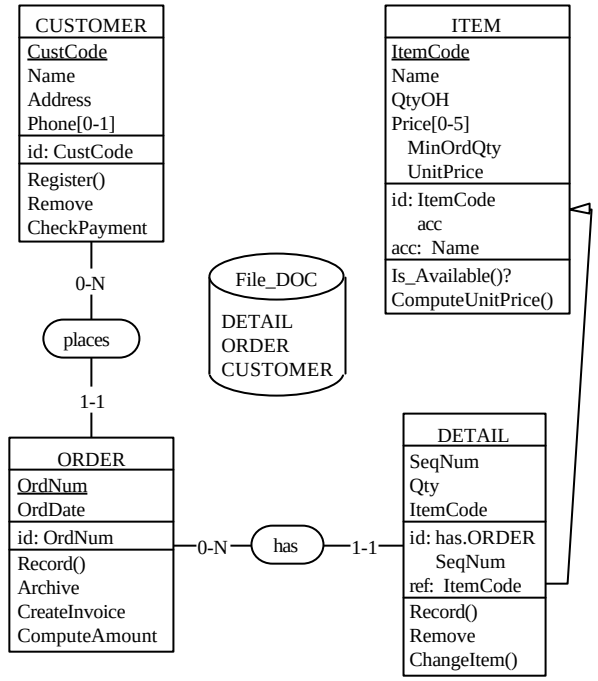

Figure 2. An illustration of the generic model. This schema includes entity types, relationship types, attributes, identifiers and processing units. It also includes foreign keys, access keys and storage spaces. This hybrid schema includes constructs from different levels of abstraction.

3.2. Model specialization

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

- *Selecting constructs.* We select the following constructs: entity types, attributes, identifiers and reference attributes.

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

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

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

For the wrapper data model and the canonical data model, the selection criteria depends on the specialized model:

- For the *wrapper data model.* We select the structures and constraints that exist explicitly in all the legacy data models since a basic wrapper must be able to keep all the structures and constraints of any underlying legacy data schema based on any data model.
- For the *canonical data model.* We define a model that is highly expressive and more flexible than the legacy data models. Such a model generally includes structures and constraints (Δ) that are unknown in the wrapper data model and hence in the legacy data models.

We can now state the relationships between these models:

$$WDM = \bigcup_i (LDM_i) \quad \text{(model wrapper)} \tag{1}$$

$$CDM = WDM + \Delta \text{ (upper wrapper)} \tag{2}$$

$$CDM = \bigcup_i (LDM_i) + \Delta \text{ (whole wrapper)} \tag{3}$$

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

$$CS = PS + \nu \text{ (instance wrapper)} \tag{4}$$

where ν is the extra semantics of CS emulated by the instance wrapper

$$ES = CS + \nu' \text{ (upper wrapper)} \tag{5}$$

where ν' is the extra semantics of ES implemented by the upper wrapper

$$\text{(4) and (5): } ES = PS + \nu + \nu' \text{ (whole wrapper)} \tag{6}$$

4. Mapping definition

To formally define the mappings, we adopt the transformational technique defined in [8]. According to this approach and the generic model, we hypothesize that producing a schema S' from a schema S' can be formalized as a schema transformation (T) and that this transformation has an inverse (T') such as:

$$S' = T(S) \tag{7}$$

$$S = T'(S') \tag{8}$$

In this way, schema transformations are essential to formally define forward and backward mappings between schemas. In addition, they can be stored in a history log that records a trace of them. The notion has been defined in [9] and can be summarized as follows.

4.1. Definition

A schema transformation is an operation that derives a target schema S' from a source S by replacing construct C (possibly empty) in S with a new construct C' (possibly empty). A transformation T can be completely defined by a couple of mappings <T,t> where T is called the structural mapping and t the instance mapping:

$$C' = T(C) \tag{9}$$

$$c' = t(c) \tag{10}$$

T explains how to replace construct C with construct C' while t states how to compute instance c' of C' from any instance c of C. Another equivalent way to describe mapping T consists of a pair of predicates <P,Q>, where P is the weakest precondition that C must satisfy for T being applicable, and Q is the strongest postcondition specifying the properties of C'. So we can also write T = <P,Q,t>.

4.2. Semantics-preserving transformation

An inverse transformation Ti can be associated with each transformation T, such that, for any construct C:

$$Pi(C) \Rightarrow C = T(Ti(C)) \tag{11}$$

T is declared semantics-preserving if the following property is preserved for any construct C and any instance c of C:

$$Pi(C) \Rightarrow C = T(Ti(C)) \wedge c = t(ti(c)) \quad (12)$$

Structural mapping T is a rewriting rule that explains how to modify the schema while instance mapping t states how to compute the instance set of C.

4.3. Semantics-preserving transformation sequence

A transformation sequence T12 = T2 o T1 is obtained by applying T2 on the schema that results from the application of T1. Such a compound transformation is semantics-preserving if its components are so. Most database engineering processes can be modeled as sequences of transformations ([1], [6]). As an illustration, Figure 3 shows a sequence of two transformations usually used in database engineering process. The first one (T1) replaces a foreign key with a relationship type and the second one (T2) expresses a multiple attribute as an external entity type.

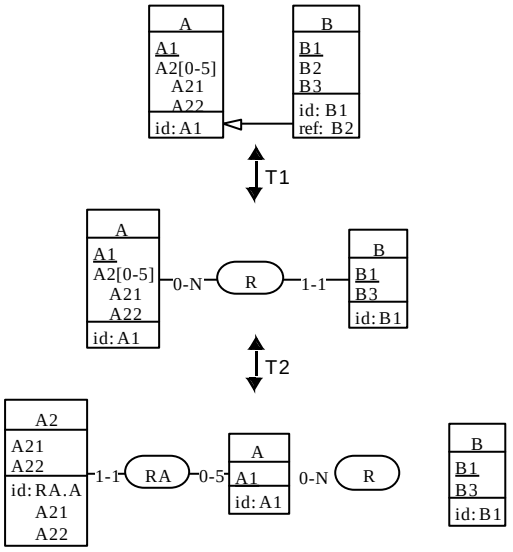

Figure 3. Sequence of two common semantics-preserving schema transformations: a foreign key transformation followed by an attribute transformation into an entity type.

Defining a component schema (CS) from the physical schema (PS) can be described as a compound semantics-preserving transformation P-to-C = <P-to-C,p-to-c> in such a way that:

$$CS = P\text{-to-}C(PS) \quad (13)$$

This transformation has an inverse: C-to-P = <C-to-P,c-to-p> such that:

$$PS = C\text{-to-}P(CS) \quad (14)$$

The physical/component mappings between the physical data and the component data can be derived from the instance mappings:

$$cs = p\text{-to-}c(ps) \quad (15)$$

$$ps = c\text{-to-}p(cs) \quad (16)$$

where ps is an instance of PS and cs an instance of CS.

Now, we are able to describe the mappings managed by the instance wrapper and the upper wrapper. An instance wrapper will rely on the mapping C-to-P to translate queries and on the mappping p-to-c to form the result instances. An upper wrapper will rely on the mapping E-to-C to translate queries and on the mapping c-to-e to form the result instances.

Of course, these mappings appear as pure functions that cannot be immediately translated into executable procedures in 3GL. However, it is fairly easy to produce procedural data conversion programs as shown in reference [8].

4.4. Transformation specialization

A set of generic transformations Γ can be defined for the generic model. These transformations can be instanciated in order to get transformations defined in a schema level (Cf. Figure 4).

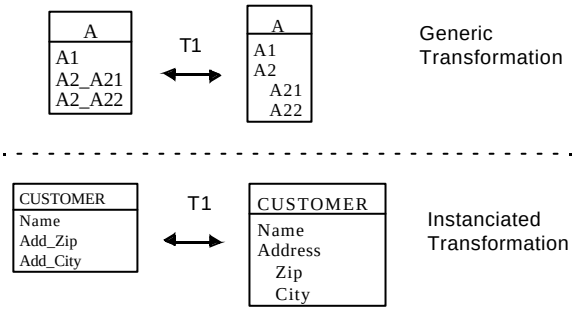

Figure 4. Representation of a generic transformation (aggregating a compound attribute) and a possible instanciation.

The production of target schema S' from source schemas S, defined in non necessarily distinct submodels, can therefore be described as a sequence of transformations from Γ. In particular, P-to-C, C-to-P, E-to-C and C-to-E can be defined by sequences of Γ transformations.

We can now refine the definition of the instance wrapper generator. It is based on a predefined set of schema

transformations Γ_{BW} defined from two models (i.e., the legacy model and the wrapper model):

$$\Gamma_{BW} \subseteq \Gamma \quad \text{and} \quad \forall T \in \text{P-to-C: T is defined in } \Gamma_{BW} \quad (17)$$

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

$$\Gamma_{UW} \ : \ \Gamma_{UW} \cup \Gamma_{BW} = \Gamma$$
$$\text{and} \quad \forall T \in \text{C-to-E: T is defined in } \Gamma_{UW} \quad (18)$$

5. Wrapper architecture

The architecture shown in Figure 5 provides a generic wrapper framework, i.e., independent of a particular legacy database and of a DMS family. It is based on the generic model described in Section 3.1.

The dark grey boxes represent the model-wrapper that is built once for a DMS family. A generator computes the light grey boxes for a particular legacy data system. They are built from the results of the reverse engineering processus that will be discussed in Section 6. Such an architecture can be instanciated in two levels:

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

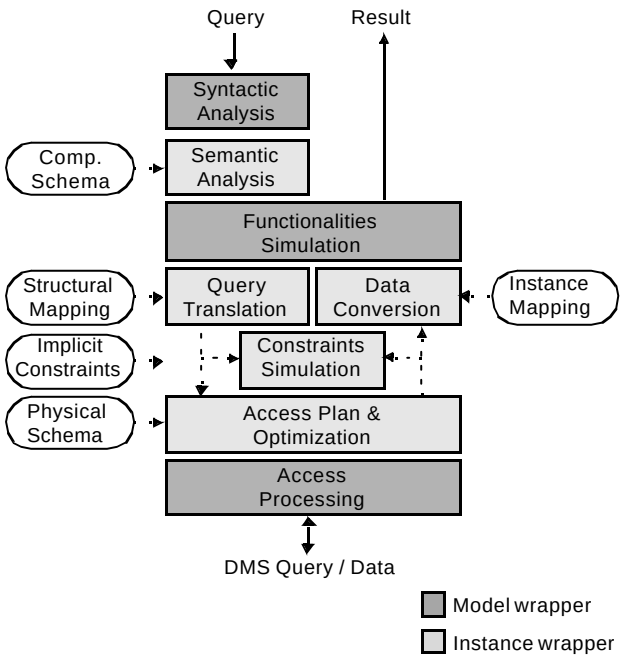

Figure 5. The components of the model and instance wrappers.

5.1. Model wrapper

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

Example 2. Let us consider a model wrapper for COBOL data sources. Such sources have limited functionalities and query capabilities (Cf. supra). The model wrapper emulates, among others: (1) syntactic analysis of the input query (through a BNF definition of the wrapper query language); (2) the cursor concept; (3) the transaction concept (if the wrapper provides update queries).

5.2. Instance wrapper

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

- the structural mappings between the physical and component schemas (C-to-P) for the **query translation** (for instance: the translation of the input query into SQL query or COBOL program codes);
- the instance mappings (p-to-c) that define the **data assembly** (for instance, building an object from a set of rows);
- the implicit constraints Δ to **simulate** (for instance, the simulation of a referential constraint);
- the structure of the data source (PS) for the **access plan and optimization** (for instance, the presence of access keys or clusters).

Example 3. Let us consider an instance wrapper for COBOL data sources. It performs, among others: (1) semantic analysis of the input query (against the component schema definition); (2) translation of an input query into COBOL program code (through the mapping definition); (3) access plan optimization (through the physical schema definition); (4) simulation of the implicit constructs and constraints (through the component schema definition).

6. Methodology for wrapper development

Considering a legacy database, building the architecture components of a wrapper is a complex engineering activity. It requires to develop the components of the model wrapper, the instance wrapper and the upper wrap-

per. In the following sections, we will describe and illustrate the processes of the methodology.

6.1. Methodology for building the model wrapper

The model wrapper is based on the definition of the wrapper query and model on the one hand, and the definition of the legacy query and model on the other hand. Except for the syntactic analysis, building wrapper components for a new model is a new problem of its own, requiring specific reasoning and techniques.

6.2. Methodology for semantics recovery

Both the instance and upper wrappers require to recover the physical, component and enriched schemas of the legacy database. They also require to define the physical/component and component/enriched mappings modeled through compound semantics preserving transformations.

Extracting a semantically rich description from a database is the main goal of the **data-centered reverse engineering process** (DBRE). A general DBRE methodology has been developed in the DB-MAIN laboratory and has been extended for building wrappers. Since it has been presented in former papers (see [7], [15] and [16]), we will only outline the main steps of the methodology.

6.2.1 Physical schema recovery

This phase consists in recovering the physical schema, including all the explicit (i.e., *declared*) structures and constraints. Database and standard files generally supply a description of this schema (data dictionary contents, DLL texts, file sections, etc.) This process consists in analyzing the data structure declaration statements included in the DDL schema scripts and application programs. It produces the physical schema based on the data model (LDM) of the legacy database.

6.2.2 Semantics recovery

This phase consists in extracting a semantically rich description from a data source. It is carried out by transforming the physical schema into the *enriched schema*. Through this process, the initial physical schema is enriched through specific analysis techniques that search non declarative sources of information for evidence of *implicit data structures and constraints*, that are progressively added to the schema. In addition, the names are translated to make them more meaningful and purely physical structures, such as indexes and storage structures, are discarded. The result of this process is the enriched schema and the transformation sequences P-to-E.

6.2.3 Component schema definition

The objective is to identify and to extract all the constructs and constraints that are not supported by the wrapper model. This extraction is performed through transformation sequence E-to-C. The result of this process is the component schema. The extracted constructs will be simulated by the upper wrapper.

6.2.4 Mappings definition

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

- the sequence emulated by the instance wrapper (P-to-C);
- the sequence not supported by the instance wrapper (C-to-E).

These sequences are built from the transformation sequences P-to-E and E-to-C got during the previous steps.

6.3. Methodology for building the instance wrapper

An instance wrapper generator will be built for one DMS family only. More precisely, an instance-wrapper generator is defined by: (1) the data model of the DMS family (LDM); (2) the wrapper model (WDM) and (3) the set of transformations it manages (Γ_{BW}).

To generate a particular instance wrapper (Figure 5), the generator relies only on results of the reverse engineering process, namely,

- the component schema (CS) for the semantic analysis;
- the structural mappings (C-to-P) for the query translation;
- the instance mappings (p-to-c) for the data conversion;
- the implicit constraints (CS - PS) for the constraint simulations;
- the physical schema (PS) for the access plan and optimization.

6.4. Methodology for building the upper wrapper

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

6.5. Small case study

To illustrate the processes described above, let us consider a wrapper defined for COBOL files and that offers an

OO interface. Through the reverse engineering process, we recover the physical schema and a semantically rich description of the COBOL files (Cf. Figure 6). During this process, the physical schema is enriched with an implicit foreign key, translated into a relationship type and an implicit constraint (the quantity of an order detail must be greater to 0). The names are translated to make them more meaningful. Finally, the physical schema is cleaned from its physical structures (access keys and files).

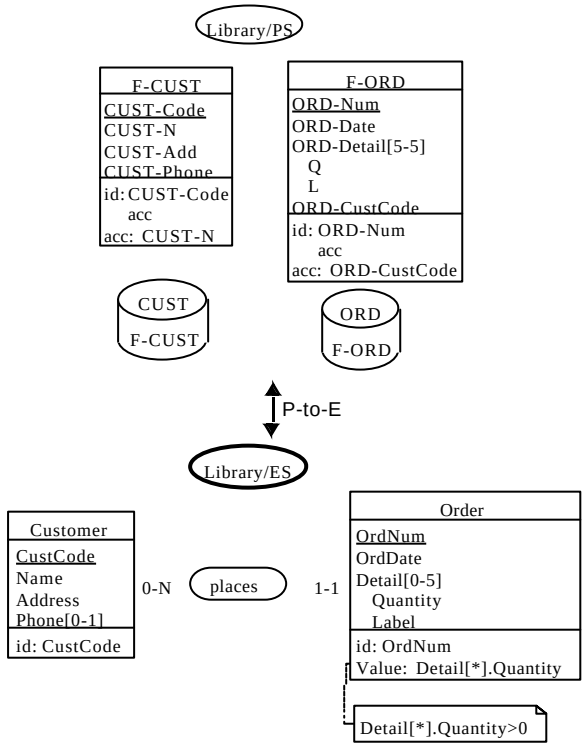

Figure 6. Physical and enriched schemas example.

Let us suppose that the wrapper model does not support a constraint such as Detail[*].Quantity>0. Consequently, the instance wrapper cannot emulate it, and it is up to the upper model to simulate it.

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

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

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

- *Implicit constraints simulation*: the cardinality [1-1] of the role played by the class Order.

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

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

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

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

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

This translation is based on the transformations (considered the reverse way) used to produce the component schema from the physical one, such as creating-relationship-type and name-conversion.

7. CASE support for wrapper development

7.1. DB-MAIN CASE tool

The DB-MAIN CASE[1] environment is dedicated to the engineering of database applications, and its scope encom-

1 An Education version of the DB-MAIN CASE environment as well as various materials of the DB-MAIN laboratory, e.g. [10], can be obtained at http://www.db-main.be

passes, but is much broader than, support for generating wrappers alone.

Besides standard functions such as specification entry, examination and management, it includes advanced processors such as transformation toolboxes, reverse engineering processors and schema analysis tools. In particular, DB-MAIN offers a rich set of semantic-preserving transformational operators that allow developers to carry out in a systematic way the physical/conceptual mapping. Another interesting feature of DB-MAIN is the meta-CASE layer, which allows method engineers to customize the tool and to add new concepts, functions, models and even new methods. In particular, DB-MAIN offers a complete development language, Voyager2, through which new functions and processors can be developed and seamlessly integrated into the tool.

In the context of wrapper building, DB-MAIN provides the developer with a toolset for reverse engineering, mappings definition, schemas specification and wrapper generation (Figure 7).

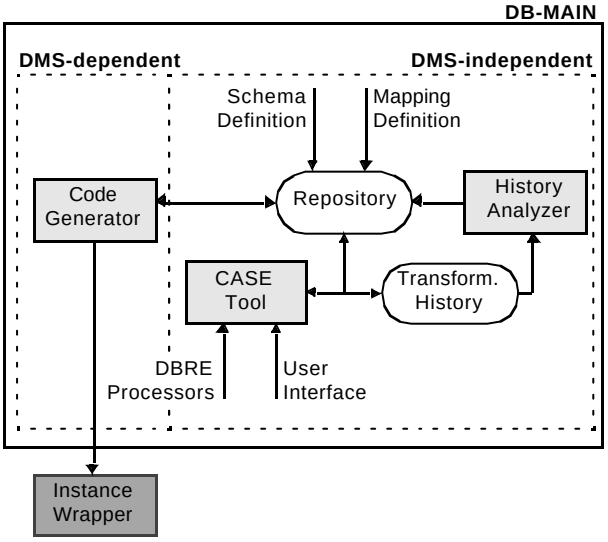

Figure 7. The extension of the DB-MAIN for building instance-wrappers.

7.2. Reverse engineering support

DB-MAIN offers functions and processors that are specific to database reverse engineering. The data structures extraction is carried out by a series of processors. 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.

Implicit construct elicitation is supported by a collection of processors that help analyze program code, schemas and data in order to find semantics. Let us mention a *dependency analyzer* that detects and displays the dependencies between the objects (variables, constants, records) of a program, a *program slicer* and a *foreign key discovery* assistant. Schema enrichment can be performed in a reliable way thanks to the semantics preserving transformation toolset. Transformation scripts that implement specific heuristics can be quickly developed. A programmable schema analysis processor can be used to detect structural patterns and problematic constructs to be further processed.

7.3. Mapping building support

DB-MAIN can automatically generate and maintain a history log (say h) of all the transformations that are applied when the developer carries out any engineering process such as data-centered analysis, optimization, implementation or reverse engineering. This history is completely formalized in such a way that it can be replayed, analyzed and transformed. For example, any history h can be inverted into history h'. Hence, if h expresses the structural mapping between the physical and component schemas, and if t is the instance mapping of h, then $\{h',t\}$ is the functional specification of the component wrapper. h' explains how to translate queries while t explains how to form the result instances. Therefore, history h can be used to generate the instance wrapper.

7.4. Instance-wrapper generation support

If $h' = C\text{-to-}P$, and if t is the instance mapping of h, that is, $t = [p\text{-to-}c]$, then $\{h',t\}$ is the mapping specification of the instance wrapper. Therefore, history h can be used to generate the instance wrapper. As suggested in Section 6, the generator is based on the description of the physical and component schemas as well as on the history of their derivation. For genericity, the wrapper generation is performed in two steps, namely the history analysis (common to all generators) and the wrapper encoding (specific to a DMS family). They have been developed in Voyager 2. At the current time, wrapper encoders for COBOL files and relational data structures are available.

8. Conclusions

In this paper, we have presented a generic architecture of data wrappers dedicated to legacy databases. We consider that a wrapper must offer a semantically rich description of its underlying source. We propose therefore a

wrapper development methodology based on data centered reverse engineering.

We have designed the wrapper architecture such that it can be instanciated for any legacy data models and systems. We propose a strong formal basis for the building of architecture components. The methodology is based on a formal transformational approach to schema engineering. This approach formally defines the mappings on which the architectural components rely. Thanks to this approach, an important part of the wrapper code can be derived in a systematic way.

The methodology is supported by an operational CASE tool, namely DB-MAIN, that gives the developer an integrated toolset for reverse engineering, inter-schema mappings definition and processing, and code generation.

Though important issues have not been tackled so far, such as optimization processing and transaction management, the architecture as well as the methods and engineering tools we have developed provide an adequate framework for wrapping legacy data systems.

At this time, we have built wrapper encoders for COBOL files and relational data structures. The wrappers offer an interface based on a pure JAVA API and a variant of the OQL language, so that they can be integrated into a great variety of architectures. We are now applying these wrapper encoders to an operational database system of a city administration.

9. References

[1] C. Batini, G. Di Battista, G. Santucci, *"Structuring Primitives for a Dictionary of Entity Relationship Data Schemas"*, in IEEE TSE, Vol. 19, No. 4, 1993.

[2] A. Bouguettaya, B. Benetallah, A. Elmagarmid, *"Interconnecting Heterogeneous Information Systems"*, Kluwer Academic Publishers, 1998.

[3] M. Brodie, M. Stonebraker, *"Migrating Legacy Systems"*, Morgan Kaufmann, 1995.

[4] 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.

[5] 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.

[6] J-L. Hainaut, C. Tonneau, M. Joris, M. Chandelon, "Schema Transformation Techniques for Database Reverse Engineering", in Proc. of the 12th Int. *Conf. on ER Approach*, E/R Institute, Arlington-Dallas, 1993.

[7] J-L. Hainaut, M. Chandelon, C. Tonneau, M. Joris, "Contribution to a Theory of Database Reverse Engineering", in Proc. of the *IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press, Baltimore, May 1993.

[8] 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.

[9] J-L. Hainaut, *"Specification Preservation in Schema Transformations - Application to Semantics and Statistics"* , Data & Knowledge Engineering, Elsevier Science Publish, 16(1), 1996.

[10] J-M. Hick, V. Englebert, J. Henrard, D. Roland, J-L. Hainaut, *"The DB-MAIN Database Engineering CASE Tool (version 6) - Functions Overview"*, DB-MAIN Technical manual, Institut d'informatique, University of Namur, 2000.

[11] E.P. Lim, H.K. Lee, *"Export Database Derivation in Object-oriented Wrappers"*, Information and Software Technology, 41, pp. 183-196, 1999.

[12] 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.

[13] C. Parent and S. Spaccapietra, "Issues and Approaches of Database Integration", *Communications of the ACM*, 41(5), pp.166-178, 1998.

[14] H.M. Sneed, "Program Interface Reengineering for Wrapping" in Proc. of *the 4rd IEEE Working Conf. on Reverse Engineering*, IEEE Computer Society Press, 1997.

[15] 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 Computer Society Press, New-York, August 1998.

[16] 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, Florence, September 1999.

[17] 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.

[18] G. Wiederhold, "Mediators in the Architecture of Future Information Systems", *IEEE Computer*, pp. 38-49, March 1992.

[19] 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.