Available online at www.sciencedirect.com

ScienceDirect

Data & Knowledge Engineering 59 (2006) 534–558

DATA & KNOWLEDGE ENGINEERING

www.elsevier.com/locate/datak

Database application evolution: A transformational approach

Jean-Marc Hick [*], Jean-Luc Hainaut

Department of Computer Science, University of Namur, Rue Grandgagnage 21, B-5000 Namur, Belgium

Received 10 October 2005; accepted 14 October 2005
Available online 21 November 2005

Abstract

While recent data management technologies, such as object oriented techniques, address the problem of database schema evolution, standard information systems currently in use raise challenging evolution problems. This paper examines database evolution from the developer point of view. It shows how requirements changes are propagated to database schemas, to data and to programs through a general strategy. This strategy requires the documentation of database design. When absent, such documentation has to be rebuilt through reverse engineering techniques. Our approach, called DB-MAIN, relies on a generic database model and on transformational paradigm that states that database engineering processes can be modeled by schema transformations. Indeed, a transformation provides both structural and instance mappings that formally define how to modify database structures and contents. We describe both the complete and a simplified approaches, and compare their merits and drawbacks. We then analyze the problem of program modification and describe a CASE tool that can assist developers in their task of system evolution. We illustrate our approach with Biomaze, a biochemical knowledge-based the database of which is rapidly evolving.
© 2005 Elsevier B.V. All rights reserved.

Keywords: Evolution; Database conversion; Schema transformation; History; Reverse engineering; CASE tools

1. Introduction

Software evolution is one of the most challenging problems in software engineering. Changes in functional or non-functional program requirements naturally induce code modification in order to make them comply with these new requirements.

A *database application* is a software system that includes complex and high-volume persistent data stored in a set of files or in a genuine database. It also has to evolve due to environment requirements changes.

Such an application brings its own specific problems, notably due to the sharing of the database among many programs. Any change in the application domain or in the technological platform of the applications induces semantic or technical adjustments in the database that, most generally, translate into structural mod-

[*] Corresponding author.
E-mail addresses: jmh@info.fundp.ac.be (J.-M. Hick), jlh@info.fundp.ac.be (J.-L. Hainaut).
URL: http://www.info.fundp.ac.be/libd (J.-M. Hick).

0169-023X/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2005.10.003

ification of the conceptual, logical, or physical schema of the database. Moreover, the data themselves must evolve accordingly, as well as the programs that read and update these data.

The lack of support, in terms of methods and tools, in database maintenance and evolution is testified by the low number of scientific references that can lead to practical application and the current state of the art among practitioners. Systematic rules for translating requirements evolution into technical modifications of an application are still unknown, particularly when traceability of the design and maintenance processes is missing. Current CASE tools automatically generate incomplete DDL code, that therefore must be modified to be truly operational. If database specifications change, these tools produce new code which is disconnected from the previous version. In addition, data conversion and program modification are up to the programmer.

Quite frustratingly, though schema evolution has been widely studied in the scientific literature, yielding interesting results, it has yet to be implemented into practical technology and methodology. The problem of database evolution was first studied for standard data structures. Several researchers have analyzed direct relational schema modification (e.g., [2,27,30,8]). Others have analyzed the propagation of conceptual modifications on relational schemas [27,32]. The object paradigm has proved to be an adequate framework to develop elegant solutions through the concepts of schema and instance versioning [1,4,25]. In [26], aspect oriented extensions of a database system allowing evolution are studied. Evolution has also been studied as a global process, independent of the underlying model. For instance [7,24] both develop a general framework in which different types of evolution are classified.

Several research projects have addressed the problem of change in information systems. The NATURE project [19] has developed a requirements engineering framework that includes modification management. The SEI-CMU project studied the evaluation of the evolution capacity of existing information systems [6]. Closer to our data-centric approach, the Varlet project [18] adopts a reverse engineering process that consists of two phases. In the first phase, the different parts of the original database are analyzed to obtain a logical schema for the implemented physical schema. In the second phase, this logical schema is transformed into a conceptual schema that is the basis for modification activities. Due to simplified conceptual schema and design process, the Varlet approach enjoys full automation, contrary to the approach presented in this paper, that allows a higher degree of freedom.

As illustrated in this paper, database evolution can be perceived as a transformational process, which complies with the emerging model-driven engineering paradigm. In [23], for instance, the authors show how a framework based on a low-level hypergraph-based data model and on a set of transformational operators provides a way to handle schema evolution in heterogeneous database architectures.

The reader will find additional material in software evolution in general and in schema evolution in particular in surveys such as [22,24,28,29].

This paper analyzes the phenomenon of data evolution in database applications as the coordinated modification of three system components, namely data structures, data and programs, following requirements changes at different levels of abstraction. We give the problem statement in Section 2 and introduce the methodological foundations in Section 3. Then, in Section 4 we describe the evolution strategy of our DB-MAIN approach.[1] We describe a representative application in Section 5, and conclude in Section 6.

This paper is an extended version of Ref. [17]. In particular, it includes a more detailed description of the transformational techniques (Sections 3.2 and 3.4), a new synthetic technique for tracing inter-schema mapping (Section 4.5) and the description of a bioinformatics knowledge-based system that integrates a database evolution engine (Section 5).

2. Problem statement

The phenomenon of evolution is analyzed in the framework of classical modeling approaches that are now familiar to database developers. These approaches consider database design to be a complex activity made up of elementary processes based on three abstraction levels, each of them dealing with homogeneous design requirements, i.e., the conceptual, logical, and physical levels. System engineering generally considers three

[1] DB-MAIN stands for *Database Maintenance and Evolution*. We introduced this approach in [10] and developed it in detail in [16].

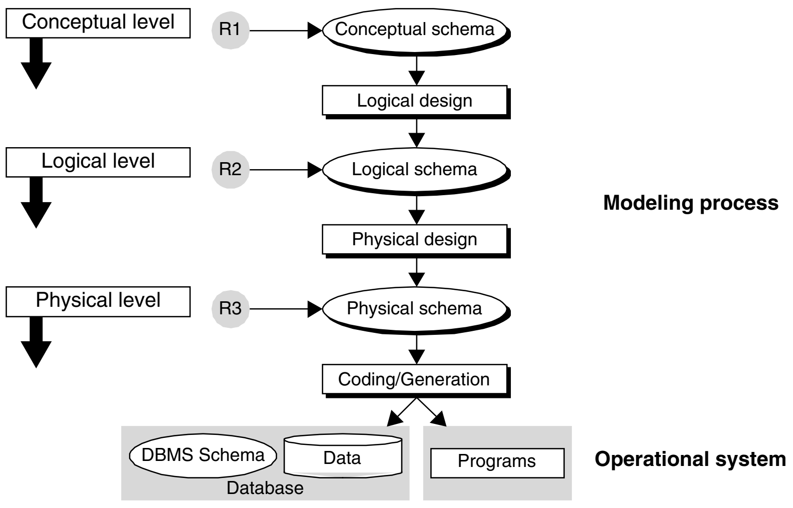


Fig. 1. Standard modeling approach distributed in three abstraction levels.

kinds of requirements, namely functional (to meet the user requirements in terms of system functions), organizational (to answer the framework changes in a company), and technical (adaptation to the new technical or hardware constraints). Fig. 1 depicts the standard database design framework, where the conceptual schema meets the organizational and functional requirements R1, while the logical schema satisfies the DBMS-dependent data model requirements R2 and the physical schema integrates the physical requirements R3. The operational system contains the database (structures and data) and the programs. According to the most commonly agreed approaches, the conceptual schema is translated into a DBMS model-dependent logical schema, which is in turn enriched into the physical schema. These translation processes are basically of a transformational nature.

We adopt the hypothesis that all the application specifications for each abstraction level and the trace of their design processes, i.e., the transformations that have been carried out, are available. This assumption is of course unrealistic in many situations, in which the program source code and DDL scripts (or DBMS data dictionary contents) are often the only available documentation. In this context, the strategies we develop must be completed to take the lack of high-level specifications into account.

Database application evolution translates requirements changes into system changes. This paper focuses on persistent data, i.e., the set of files or databases that store information that describes the application domain. More precisely, the problem can be summarized as follows: how must a change in a schema be propagated to (1) the lower level schemas, including DDL code, (2) the data and (3) the application programs.

This study relies on a *non-temporal* approach, according to which all the application components (i.e., the schemas, data and programs) are replaced by new versions. In particular, the data are transformed in such a way that they become unavailable in their previous form. Application programs can use the new data only after being transformed accordingly. This strategy applies both to legacy and modern database applications and contrasts with advanced systems where the modification of the database schema is translated into the addition of a new version that coexists with former versions, possibly through views and filtering mechanisms. In such schema/data versioning approaches [20,27], the old schema is preserved and access to its data is stored or calculated.

For example, removing a property from a conceptual class is ultimately translated in our approach into the removal of the corresponding column through the query:

```
alter table < table_name > drop < column_name > .
```

3. Methodological foundations

Requirements modification is translated into specification changes at the corresponding level, according to rules that we do not address in this paper. In an ideal context in which all the schemas are available and consistent, these changes must be propagated upward and downward to other abstraction levels. Due to the complexity of the process, it must be supported by a methodology and a CASE tool, which must meet three conditions.

- *Genericity*: it must offer a generic model of specification representation whatever the abstraction level, the technology, or the paradigm on which the application relies. Indeed, most operational environments include several technologies and rely on several models.
- *Formality*: it must formally describe database engineering activities in order to allow rigorous reasoning, notably as far as specification preservation is ensured.
- *Traceability*: the links between specifications must be precisely recorded. They must be analyzed to provide the information necessary for modification propagation.

The DB-MAIN approach to database evolution is based on three concepts that implement these requirements: a generic representation model (Section 3.1), a transformational approach (Section 3.2), and history management (Section 3.4).

3.1. Generic model of specification representation

The DB-MAIN model has generic characteristics according to two dimensions:

- specification representation at each abstraction level: conceptual, logical, and physical;
- coverage of the main modeling paradigms or technologies such as ERA, UML, ORM, objects, relational, CODASYL, IMS, standard files, and XML models.

It is based on the generic Entity/Relationship model and supports all the operational models through a specialization mechanism. Each model is defined as a sub-model of the generic model, obtained by restriction, i.e., by selecting the relevant objects, by defining the legal assemblies through structural predicates, by renaming the object according to the model taxonomy, and by choosing a suitable graphical representation. Fig. 2 presents schemas according to classical sub-models for the three abstraction levels: ERA (European style) at the conceptual level, relational at the logical level, and Oracle 8 at the physical level.

In Fig. 2/C, *PERSON*, *CUSTOMER*, *SUPPLIER*, *ORDER* and *ITEM* are entity types or object classes. *CUSTOMER* and *SUPPLIER* are subtypes of *PERSON*, which acts as a supertype. Totality and disjunction constraints (P denotes both, and defines a partition) are defined on these subtypes. Attributes *PersNum*, *Name*, *Address* and *Phone* characterize *PERSON* (as well as *CUSTOMER* and *SUPPLIER*). *Address* is a compound attribute while *Phone* is multivalued. Attributes *Number*, *Street* and *City* are components of *Address*. *Number* is optional. Relationship types (rel-types for short) *place*, *reference* and *offer* are binary relationship types. Rel-type *reference* has an attribute. *ORDER* plays two roles, respectively in *place* and *reference*. Each role has minimal and maximal cardinalities (N stands for *infinity*), that are noted in the *participation* interpretation, which is the converse of the *look-across* interpretation of, say, UML. Rel-type *reference* is qualified many-to-many and *place* one-to-many. *CUSTOMER* is identified by *CusNum*.

Fig. 2/L depicts a relational schema in which *PERSON*, *CUSTOMER*, *SUPPLIER*, *PHONE*, *ORDER*, etc. are tables. *PersNum*, *CusNum* and *Name* are columns of *PERSON*. *Name* is mandatory and *Add_Number* is optional (nullable). *PERSON* has a primary identifier (primary key), *NumPers* and two secondary identifiers, *CusNum* and *SupNum*. *ORDER.CusNum*, as well as *PERSON.CusNum*, are *foreign keys* (*ref* or *equ*) targeting *CUSTOMER*. All the values of *CUSTOMER.CusNum* also appear as non-null values of *PERSON.CusNum*. *PERSON* is subject to an *exactly-one* constraint (*exact-1*), i.e., for each row of this table, one and only one of the columns *SUPPLIER* and *CUSTOMER* has a non-null value.

In Fig. 2/P, the names of tables and columns are compliant with SQL syntax and include physical, performance-oriented constructs. For example, *Date* (reserved word) becomes *DATEORD* in *ORDER*. Indexes (*acc*ess keys) are defined on columns such as *PERSNUM* of *PERSON* and *SUPNUM* of *ITEM*. Storage spaces (called TABLESPACE in Oracle) are defined: *SPC_SUP* contains the rows of tables *ITEM* and *SUPPLIER*.

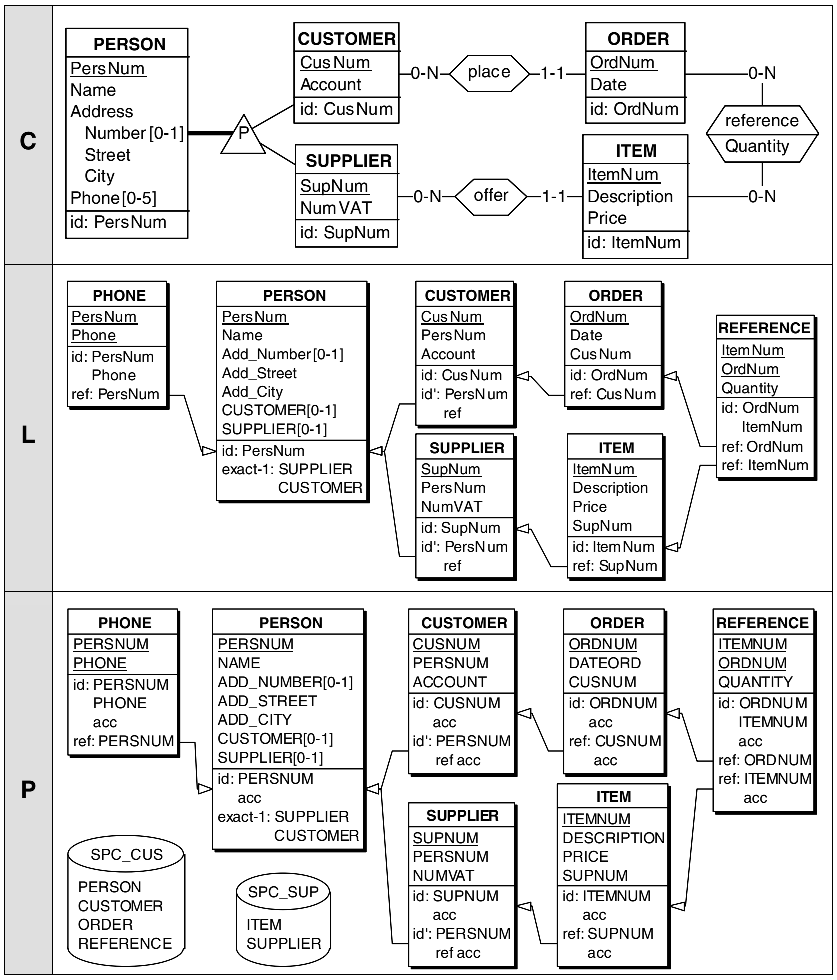


Fig. 2. Graphical views of conceptual (C), logical (L), and physical (P) schemas.

3.2. Transformational approach

A database engineering process can be defined as a sequence of *data structure transformations* [3]. Adding an entity type, renaming an attribute, and translating a relationship type into a foreign key are elementary transformations. They can be combined to build more complex processes such as schema normalization, logical schema optimization, or DDL code generation. The concept of transformation used in this paper is described formally in [11,13], but we will briefly present those of its principles that are of interest in this paper.

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, a transformation Σ is defined as a pair of mappings $\langle T, t\rangle$ such that:

$$C' = T(C),$$
$$c' = t(c),$$

where c is any instance of C, and c' is the corresponding instance of C'.

Structural mapping T explains how to modify the schema while instance mapping t states how to compute the instance set of C' from the instances of C (Fig. 3). Structural mapping T can be defined by a pair of predicates $\langle P, Q\rangle$ where P is the minimal precondition C must satisfy and Q the maximal postcondition observed in

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

Fig. 3. General transformation pattern.

C'. Therefore, we can alternatively use the notations $\Sigma = \langle T, t\rangle$ and $\Sigma = \langle P, Q, t\rangle$. P (resp. Q) is a second-order predicate that defines the properties of structure C (resp. C').

Any transformation Σ can be given an inverse transformation $\Sigma' = \langle T', t'\rangle$ such that

$$T'(T(C)) = C.$$

If, in addition, we also are provided with instance mapping t' such that:

$$t'(t(c)) = c,$$

then Σ (and Σ') are said semantics-preserving or *reversible*. If $\langle T', t'\rangle$ is also reversible, Σ and Σ' are called *symmetrically reversible*.

Fig. 4 illustrates the structural mapping T of the transformation of a relationship type into an entity type. Both left and right sides are graphical representations of predicates P and Q. This classical transformation appears in logical design, where complex structures, such as n-ary or many-to-many rel-types must be replaced with simpler, flat structures. This transformation is abstract, or generic, in that the concerned schema objects are denoted by variables A, B (for entity type names), r (rel-type name), rA and rB (role names). Assigning names of actual object leads to a concrete, or instantiated transformation such as that of Fig. 5. The instance mapping, which is not represented in Fig. 5, explains how each R entity derives from an r relationship. The inverse transformation, denoted by T' in Fig. 4, transforms the entity type R into the rel-type r. A complete formal description of this transformation, including the proof of its symmetrical reversibility, can be found in [11].

In the context of the source schema, a transformation is specified by its signature, which gives the name of the transformation, the name of the objects concerned in the source schema and the name of the new objects in the target schema. For example, the signatures of the generic transformations represented in Fig. 4 are:

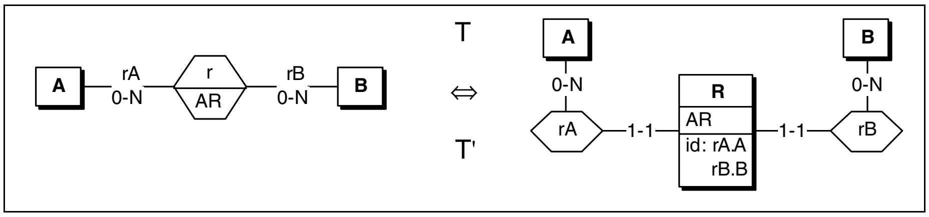


Fig. 4. Two, inverse, generic transformations through which relationship type r is transformed into entity type R (T) and conversely (T').

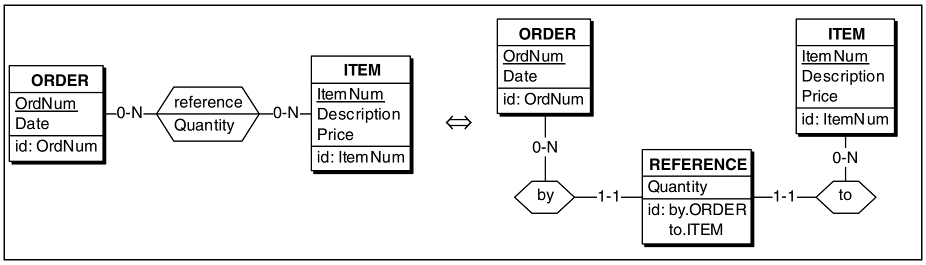


Fig. 5. Instantiation of the generic transformations of Fig. 4 with actual object names.

$$\mathrm{T} : (\mathrm{R}, \{\mathrm{rA}, \mathrm{rB}\}) \leftarrow \texttt{RT-to-ET(r)}$$
$$\mathrm{T}' : \mathrm{r} \leftarrow \texttt{ET-to-RT(R)}$$

The first expression reads as follows: by application of the RT-to-ET transformation on rel-type *r*, a new entity type *R* and two new rel-types *rA* and *rB* are created. Note that all objects need not be mentioned in a signature. Such is the case, in the *T*′ signature, of rel-types *rA* and *rB* in which *R* participates.

The notion of *semantics* of a schema is an important issue, though it has no generally agreed upon definition. We assume that the semantics of S1 include the semantics of S2 *iff 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 an entity type).
- *T*− includes the transformations that decrease the semantics of the schema (for example adding an identifier).
- *T*= is the category of transformations that preserve the semantics of the schema (for example the transformation of Fig. 4).

Transformations in *T*= are mainly used in logical and physical schema production, while *T*+ and *T*− transformations make up the basis of specification and evolution processes.

3.3. Practical transformations

T+ and *T*− transformations are fairly straightforward, but *T*= operators deserve to be further described. In this section, we illustrate some of the most useful semantics-preserving transformations. In particular, they are necessary and sufficient to transform the conceptual schema of Fig. 2/C into a relational schema (Fig. 2/L). Fig. 6 shows graphically the source (*P*) and target (*Q*) patterns of these operators. All of them are symmetrically reversible, so that their use is guaranteed to produce target schemas that preserve the semantics of a large class of source conceptual schemas.

Transformation `ISA-to-RT` replaces a supertype–subtype pattern by one-to-one rel-types between the supertype and each subtype. A more general version of transformation `RT-to-ET` has been described in Fig. 4. Transformation `Att-to-ET-inst` replaces an attribute by an entity type and a one-to-many rel-type. Transformation disaggregate replaces a compound attribute by its components. Finally, transformation `RT-to-FK` replaces a one-to-many rel-type by a foreign key.

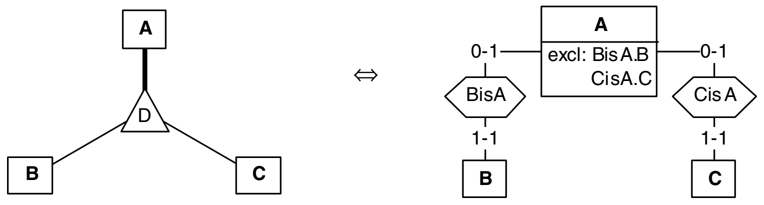
3.4. History

For the sake of consistency, we consider that requirements modifications applied at a given abstraction level must be propagated to the other levels. For example, adding a column to a table must imply the addition of the corresponding attribute to the entity type implemented by this table. Conversely, removing a one-to-many rel-type in a conceptual schema must be followed by the removal of the corresponding foreign key in the logical and physical schemas. As far as evolution is concerned, keeping track of the design transformations is a necessity, as we will see, to avoid manual reformulation of the design transformation sequence for each evolution modification [10].

The trace of the transformations that produce the schema *Sj* from schema *Si* is called the *history* of the transformation process, and is denoted as *Hij*. The composition of a sequence of elementary transformations *Hij* is also a (macro-)transformation, so that we can use the functional notation: $S_j = H_{ij}(S_i)$ with $H_{ij} = T_n \circ \cdots \circ T_2 \circ T_1$, that will be denoted as $\langle T_1, T_2, \ldots, T_n \rangle$ in the following.

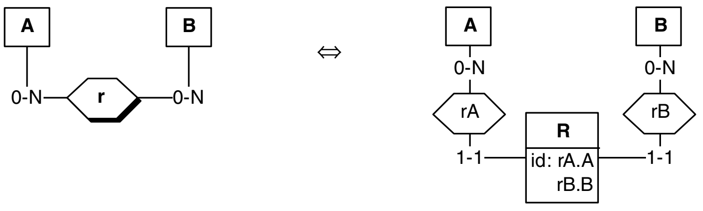
Using the signature notation, the following history, named *LD0*, describes how the conceptual schema of Fig. 2/C has been transformed into the relational schema of Fig. 2/L.

Σ1



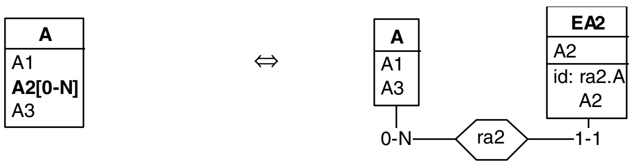
(Bisa,Cisa) ← ISA-to-RT(A,{B,C})
() ← RT-to-ISA(A,{Bisa,Cisa})

Σ2



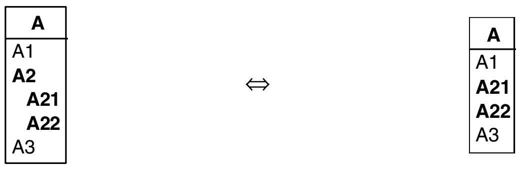
(R,{rA,rB}) ← RT-to-ET(r)
(r) ← ET-to-RT(R)

Σ3



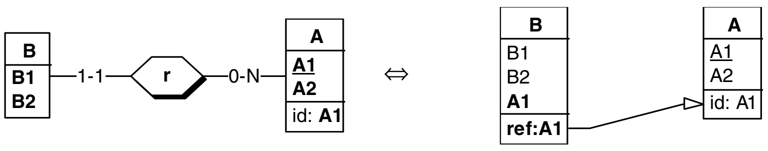
(EA2,rA) ← Att-to-ET-inst(A.A2)
(A2) ← ET-to-Att(EA2)

Σ4



(A21,A22) ← disaggregate(A.A2)
(A2) ← aggregate(A.A21,A.A22)

Σ5



(A1) ← RT-to-FK(B,r)
(r) ← FK-to-RT(B.A1)

Fig. 6. Five of the most common generic transformations (left to right) and their inverse (right to left), together with their respective signatures.

```
LDO = <
  T1:(pers_cus,pers_sup) ← ISA-to-RT(PERSON, {CUSTOMER,SUPPLIER})
  T2:(REFERENCE, {ord_ref,ite_ref}) ← RT-to-ET(reference)
  T3:(PHONE,have) ← Att-to-ET-inst(PERSON.Phone)
  T4:(Add_Number,Add_Street,Add_City) ← disaggregate(PERSON.Address)
  T5:(CUSTOMER,CusNum) ← RT-to-FK(pers_cus)
  T6:(SUPPLIER,SupNum) ← RT-to-FK(pers_sup)
  T7:(PHONE,PersNum) ← RT-to-FK(have)
  T8:(ORDER,CusNum) ← RT-to-FK(place)
  T9:(ITEM,SupNum) ← RT-to-FK(offer)
  T10:(REFERENCE,OrdNum) ← RT-to-FK(ord_ref)
  T11:(REFERENCE,ItemNum) ← RT-to-FK(ite_ref)
>
```

When the transformations recorded in H_{ij} are applied to S_i, H_{ij} is said to be *replayed* on S_i. A history can be formally processed if several properties hold [12]:

- *Exhaustivity*: the transformations are recorded precisely and completely to allow the inversion of non semantics-preserving transformations (reversing a delete transformation requires a description of all the aspects of the deleted objects).
- *Normalization*: the history is monotonic (no rollback) and linear (no multiple branches).
- *Non-competition*: a history is attached to one schema and only one user can modify it.

An important manipulation is the *inversion* of a history. Considering history H, it consists in computing the inverse history H' such that, for source schema S, we have:

$$S = H'(H(S))$$

For linear histories, the inverse is obtained by replacing each transformation with its inverse, then reversing the order of the transformations. The reconstruction of the design history of a legacy database from the history of its reverse engineering is one of the applications of this operation. It is one of the bases of the strategies we describe in Section 4.

4. Evolution strategy

The approach proposed in this paper has been developed for controlling the evolution of relational databases and of the client applications that use them, but other models can be accommodated with minimal effort thanks to the genericity of the transformational approach. This choice allows us to build a modification typology and to design concrete conversion tools that can be used with systems developed in a third-generation language such as COBOL/SQL or C/SQL.

To make database applications evolve, the design history must be available. In particular, the three levels of specification must exist and be documented, together with the histories of the inter-level transformation processes. In other words, the database is required to be fully documented through its conceptual, logical, and physical schemas, and through the histories of the conceptual-to-logical and logical-to-physical processes. In most cases, these preconditions are not met: some (or all) levels are missing, incomplete or obsolete. Sometimes, only the source code of the programs and of the data structures are available. In these cases, the documentation and the histories must be rebuilt using reverse engineering techniques that are not addressed in this paper. The reverse engineering approach we have developed aims at rebuilding not only the three schemas of a legacy database, but also the histories of their production. It has been described in [9,14] while the process of rebuilding histories has been developed in [12].

4.1. Evolution scenarios

We have defined three scenarios, one for each abstraction level (Fig. 7). The initial specifications are available as the three standard schemas: conceptual (CS0), logical (LS0) and physical (PS0) schemas. The operational components are the database (D0, including data and structures) and the programs (P0). LD0 (resp. PD0) is the history that describes the transformations applied to CS0 (resp. LS0) to obtain LS0 (resp. PS0). The scenarios are constrained by the following assumption: the change must be applied on the relevant level. For instance, adding a new property to an application class must be translated into the addition of an attribute to a conceptual entity type, in CS0, and not by adding a column to a table in LS0, or worse, by altering the SQL code, through an *alter table* SQL statement.

In the first scenario (Fig. 7a), the modifications translate changes in the functional requirement into conceptual schema updates. This new state is called CS1. The problem we address is the propagation of these modifications towards the logical, physical, and operational layers, leading to the new components LS1, PS1, P1, and D1, and to the revised histories LS1 and PS1. The second scenario (Fig. 7b) addresses logical schema modifications. Though the conceptual schema is kept unchanged (CS1 = CS0), the logical design history LD0 must be updated as LD1 and the modifications must be propagated in the physical and operational layers. In the third scenario (Fig. 7c), the designer modifies the physical schema to meet, e.g., new performance requirements. The physical design history is updated and the operational layer is converted.

The evolution strategy comprises four steps: *database schema modification* (Section 4.2), *schema modification propagation* (Section 4.3), *database conversion* (Section 4.4) and *program modification* (Section 4.6).

4.2. Database schema modification

In Fig. 7a, the requirements met by CS0 evolve from R1 to R1′. The analyst copies the schema CS0 into CS1 and translates the changes into modifications of CS1. The transformations applied to CS0 to obtain CS1 are recorded into history CE1, so that CS1 = CE1(CS0). In Fig. 7b, the schema LS0 is modified to obtain LS1. The logical evolution transformations are recorded in the history LE1. Note that the designer can also modify the logical schema by using transformations other than those used in LD0, without modifying the conceptual objects. For example, a multivalued attribute transformed into a series of single-valued columns will now be transformed into an autonomous table. In this case, though the conceptual schema does not change, the first scenario is used. CE1 is empty and CS1 is equivalent to CS0, but the logical design history LD1 contains the new transformation (cf. Section 4.3). In Fig. 7c, the physical schema PS0 is transformed into PS1 and the modification transformations are recorded in PE1.

Our DB-MAIN approach is based on a set of standard schema modifications that account for most evolution needs. A detailed study of modification typology in conceptual, logical, and physical levels is given in [16,27].

Hereafter, we discuss the first scenario applied to the following change: the cardinality of the multivalued attribute Phone in the conceptual schema of Fig. 2/C, is changed from [0–5] to [0–2]. This example will be analyzed for each step of the evolution strategy process.

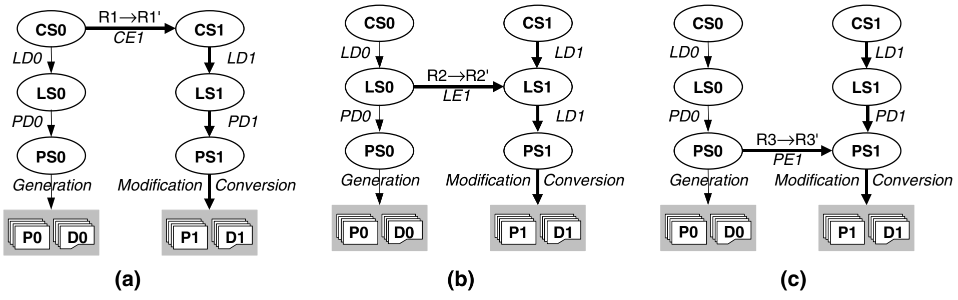


Fig. 7. Propagation of modifications at each abstraction level. (a) Conceptual modifications, (b) logical modifications, and (c) physical modifications.

The conceptual evolution history CE1 comprises one signature:

$$\mathtt{CE1} = < \mathtt{T1} : \mathtt{change\text{-}max\text{-}card(PERSON.Phone, 2)} >$$

4.3. Schema modification propagation

At this level, all the specifications and the histories must be updated according to the modification of attribute *Phone* described in Section 4.2.

In the *first scenario*, the new conceptual schema CS1 is transformed into a logical schema LS1 that is as close as possible to the former version LS0 and that integrates the conceptual modifications. Technically speaking, and in accordance with the idea that a transformation progressively changes (and therefore *destroys*) the source schema, we proceed as follows:

1. we create a new schema that is a copy of CS1, and we name it LS1;
2. we replay the history LD0 on LS1.

This history contains the necessary operations to transform the conceptual schema CS0 into a relational schema, as in Fig. 2/L. The relational transformations belong essentially to the $T=$ category, since the logical schema is expected to preserve the conceptual specifications.

When the history LD0 is replayed on the copy LS1 of the new conceptual schema CS1, four situations can occur according to the status of the object submitted to a transformation of LD0.

1. The object has not been changed by CE1: the transformations of LD0 concerning this object are executed without modification.
2. The object is new, and has been created by CE1: the transformations of LD0 have no effect. The designer must specifically process the new object.
3. The object has been removed by CE1: the transformations of LD0 concerning this object can be applied but they have no effect.
4. The object has been modified by CE1: for minor modifications (e.g., data type), the transformations concerning this object are executed. For major modifications (e.g., maximum cardinality), these transformations are no longer adapted and processing this object is the designer's responsibility.

The transformations of LD0 augmented with those applied on new objects (type 2), on modified objects (type 4) and where transformations on removed objects have been discarded (type 3) make up the new logical design process recorded as history LD1.

After that, we proceed in the same way by replaying history PD0 on LS1 to obtain the new physical schema PS1. PD1 contains the transformations (on physical structures: indexes and storage spaces) such that PS1 = PD1(LS1).

In the *second scenario*, the propagation starts at the logical level but is based on similar principles. The new logical design history LD1 is made up of LD0 augmented by the trace of the new transformations (LE1): LD1 = LD0 $\circ$ LE1. The propagation of the modifications to the physical level is similar to that of the first scenario. If the evolution involves the application of transformations other than those used formerly to produce the logical schema from the conceptual specifications, then LS0 is replayed step by step and the designer replaces, when needed, the former transformations by the new ones. A second set of schema modifications is needed: it includes the most useful conceptual-to-logical transformations for relational schemas [16].

The *third scenario* is similar to the second one, applied on the physical schema. The new physical design history PD1 is made up of PD0 augmented by the trace of the new transformations (PE1): PD1 = PD0 $\circ$ PE1. The physical design only deals with constructs such as indexes and storage spaces, which in most cases is sufficient to describe physical design and tuning scenarios. The logical design stays unchanged (LD1 = LD0).

Let us go back to our working example, in which we consider the following scenario. When replaying the history LD0 (Section 3.4) on the new conceptual schema CS1, the designer decides to change his/her mind.

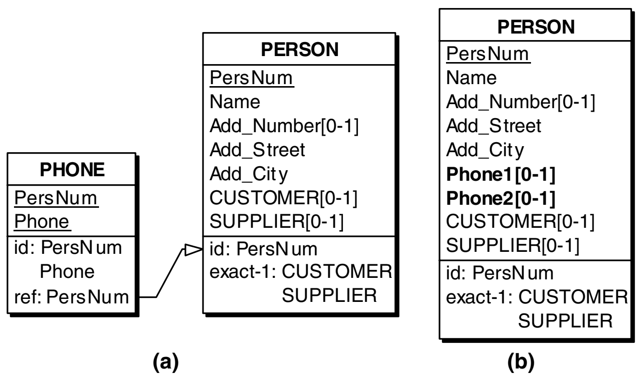


Fig. 8. Two popular representations of multivalued attribute *Phone*. (a) *Phone* represented by a table in LSO and (b) *Phone* represented by two columns in LS1.

Instead of transforming the attribute *Phone* into an entity type that ultimately becomes the table *PHONE* (Fig. 8a), s/he applies the *instantiation* transformation, according to which a single-valued attribute is introduced to store each source value. Such a design decision, which can be disputed as far as normalization is concerned, can be practically justified by the small number of values of *Phone* for each *PERSON* entity. *Phone* is therefore replaced with two optional single-valued attributes: *Phone1* and *Phone2* (Fig. 8b). The new logical design history LD1 is equivalent to LD0 except that the signature T3 is replaced with

$$\texttt{(PERSON.Phone1,PERSON.Phone2)} \leftarrow \texttt{instantiate(PERSON.Phone)}$$

and the signature T7 is removed. By simply replaying PD0 on the new logical schema, a new physical design history PD1 is obtained, in which the indexes and storage space specifications related to the former *PHONE* table are automatically skipped.

4.4. Database conversion

Once the specifications have been updated, the database structure and contents D0 can be converted. The data conversion can be automated thanks to a converter generator. The study of the evolution histories (CE1, LE1, or PE1) allows the generator to locate the modifications in the three abstraction levels. A differential analysis of the design histories (LD0, LD1, PD0, and PD1) gives the information to derive the physical structures that have been removed, modified, or added. To shorten the notation, let us call E any evolution history, and C0 and C1 any elementary or composed history in the old and in the new branch.

In the first scenario, we have:

$$\mathrm{PS0} = \mathrm{PD0}(\mathrm{LD0}(\mathrm{CS0})) = \mathrm{C0}(\mathrm{CS0}) \text{ and } \mathrm{PS1} = \mathrm{PD1}(\mathrm{LD1}(\mathrm{CS1})) = \mathrm{C1}(\mathrm{CS1}).$$

In the second scenario, we have:

$$\mathrm{PS0} = \mathrm{PD0}(\mathrm{LS0}) = \mathrm{C0}(\mathrm{LS0}) \text{ and } \mathrm{PS1} = \mathrm{PD1}(\mathrm{LS1}) = \mathrm{C1}(\mathrm{LS1}).$$

In the third scenario, C0 and C1 are empty.

According to a definite type of modification appearing in E, three distinct behaviors are possible:

1. In the case of object creation, the analysis of C1 gives the new physical structures according to the possible transformations applied successively on this object. This object has no instances.
2. In the case of object destruction, the analysis of C0 provides the old physical structures to remove, together with its instances, according to the possible transformations applied on this object.
3. The case of object modification is more complex. Initially, new structures are created on the basis of C1. Then, the data instances are transferred from the old structures to the new ones. And finally, the old physical structures are removed according to the analysis of C0.

The analysis of E, C0, and C1 let us derive the transformation of data structures and instances of PS0 (i.e., D0) into structures and instances of PS1 (D1). The chain of structural mappings (*T* parts) drives the schema modification while the chain of instance mappings (*t* parts) defines the way data have to be transformed. These transformations are translated into a converter made up of SQL scripts or programs[2] in more complex cases. The detail of translation of schema transformations into SQL scripts can be found in [16].

Some modifications bring about information losses and constraint violations. In these cases, the conversion script produces a query to verify the violation of the constraint and to ensure data consistency. For example, if the designer creates a primary identifier on a previously non-unique column, the table contents may violate the uniqueness constraint, so that the script must store the inconsistent rows, as well as the possible dependent rows in other tables, in temporary *error* tables. Automating the generation of conversion scripts is always possible, but user intervention is often necessary to correctly manage ambiguous or conflicting instances.

In our scenario, the analysis of CE1, C0, and C1 shows that the old table *PHONE* in source schema PS0 must be replaced by the columns *PHONE1* and *PHONE2* in target schema PS1. A converter translates this transformation into the following Oracle script, which converts both the database structure and contents, and which stores the conflicting data, if any, into the table *TEL_ERROR*.

```
– Creation of columns Phone1 and Phone2
ALTER TABLE PERSON ADD PHONE1 CHAR(12);
ALTER TABLE PERSON ADD PHONE2 CHAR(12);


– Creation of error table TEL_ERROR
CREATE TABLE TEL_ERROR(NUMPERS INT, PHONE CHAR(12));


– Transfer of data
CREATE OR REPLACE PROCEDURE Trf_data IS
  CURSOR c1 IS SELECT * FROM PERSON P
    WHERE exists(select * from PHONE where NUMPERS = P.NUMPERS);
  CURSOR c2 IS SELECT * FROM PHONE where NUMPERS = num;
  tP c1%ROWTYPE; tT c2%ROWTYPE; num INT; comp NUMBER;
  BEGIN
    FOR tP IN c1 LOOP
      comp := 1; num := tP.NUMPERS;
      FOR tT IN c2 LOOP
        IF comp = 1 THEN
          UPDATE PERSON SET PHONE1 = tT.PHONE WHERE NUMPERS = tP.NUMPERS; END IF;
        IF comp = 2 THEN
          UPDATE PERSON SET PHONE2 = tT.PHONE WHERE NUMPERS = tP.NUMPERS; END IF;
        IF comp > 2 THEN
          INSERT INTO TEL_ERROR VALUES(tP.NUMPERS,tT.PHONE); END IF;
        comp := comp + 1;
      END LOOP;
    END LOOP;
  END;


  – PHONE deletion
  DROP TABLE PHONE CASCADE CONSTRAINT;
```

[2] Such a converter is a variant of Extract-Transform-Load (ETL) processors.

4.5. Synthetic approach

The previous section describes a general approach based on history analysis that gives information to convert data structures and instances. The implementation of the analyzer that supports this approach is feasible, as demonstrated in [16], but complex for three major reasons:

- In practice, histories are difficult to parse because they include much information that is needed, among others reasons for undoing actions and for reversing histories [12]. For example, when deleting an entity type, information about its attributes and constraints is also recorded. This information, which is useless for database conversion, may make the history analysis process complex.
- The history analyzer is not easy to maintain when a new type of modification must be taken into account. In the industry, authorized modifications depend on the context (organizational, functional, and technical) and on the application domain. The evolution processor must be easy to update in order to be accepted by database engineers and to be financially worthwhile.
- A design history may be non-linear and may include long chains of consecutive transformations on a given object. This characteristic derives from the exploratory nature of engineering processes when they are manual or semi-automated. Such histories must be cleaned and normalized in order to be useable [12].

Moreover, field studies show that most modifications are fairly simple and straightforward. Renaming objects, merging and splitting entity types, adding an entity type, an attribute, or a relationship type, changing the domain or the cardinality of an attribute, and changing the cardinality of a relationship type account for the most commonly requested changes. Changing the implementation of a conceptual construct is rare, and most often makes use of a limited set of well defined techniques.

Hence, we have the idea of representing the inter-schema mapping through a static many-to-many relation from the constructs of one of the schemas to those of the other. In particular, we can drop the information on how exactly the target schema was produced, thus leading to a coarse, but efficient, traceability.

Basically, each source object includes an identifier that is transmitted to all the target objects that derive, directly or indirectly, from it. By examining the source and target objects that share the same identifier, the analyzer infers which transformations have been applied. Since object names can change, possibly several times in a modification session, they cannot be used to support the traceability of the evolution. Therefore, the required identifier is materialized by a temporary abstract unique *stamp* associated with each modified object.

The simplified evolution process relies on the following principles.

1. The evolution is based on the comparison of the source and target *physical* schemas, whatever the abstraction level to which the schema modification has been carried out.
2. Before the current evolution process, each object of the source schema receives a *proper stamp*, that is, a stamp that is not inherited.
3. The way in which the source schema is transformed into the target schema is ignored. In particular, all the strategies, ranging from primitive *physical-to-physical* (directly modifying the physical schema) to sophisticated *physical-to-conceptual-to-physical* (Section 4) can be applied.
4. In the schema modification and propagation phases, the transformations applied on stamped objects propagate this information to the resulting objects in the form of *inherited stamps*. The inherited stamp is a copy of the proper stamp if the source object has a proper stamp only. If the source object has received inherited stamps through a previous transformation, then the inherited stamps are copied instead. If one object is transformed into several objects (as is the case for the instantiation transformation of a multivalued attribute into a list of single-valued attributes, Fig. 8b), the target objects get the same inherited stamp. If several objects are transformed into one object, the new object inherits the list of the proper or inherited stamps of the source objects.

The evolution analyzer deduces the transformations that have been applied, and therefore the structure and data modifications to be applied, from a structural comparison between the source and target constructs that share the same stamps. Applied to the comparison of the physical schemas PS0 and PS1, the analysis algorithm builds the modification list as follows:

- For each object of the target schema PS1:
  If this object is missing in the source schema PS0, a creation operation is stored in the modification list.
  If this object exists in PS0 with the same properties (name, type, cardinalities,...), there is no modification and the object is skipped.
  If this object exists in PS0 with other properties, such as name or cardinalities, a modification operation is stored to reflect the structural change of this object.
  If this object exists in PS0 but with another type or in multiple instances, the transformations applied are inferred and are stored in the modification list.
- For each object of the source schema PS0:
  If this object is missing in the target schema, a deletion operation is stored in the modification list.

Fig. 9 illustrates the propagation of the proper stamp st:1254 of the rel-type *has* through two successive transformation steps. In the first step (top), this stamp is inherited by entity type *HAS* and rel-types *hc* and *hp* (st′:1254). In the second step (bottom), the inherited stamps of these rel-types are propagated to the equivalent foreign keys *Cust#* and *Phone#* of *HAS*.

The evolution analyzer identifies all the objects with stamps 1254, that is, rel-type *has* in the source schema, and entity type *HAS* and rel-types *hc* and *hp* in the target schema. This mapping clearly identifies the standard chain of transformations of a many-to-many rel-type into relational structures.

This technique is not deterministic in two situations.

First, the analyzer may derive more than one possible transformation. In Fig. 10, the attribute *CUSTOMER.Phone* of CS1 becomes multivalued. The comparison between PS0 and PS1 gives two possible interpretations of the conceptual modification:

- *Phone* has the new type char(30) in place of char(15).
- *Phone* is the aggregation of two phone numbers.

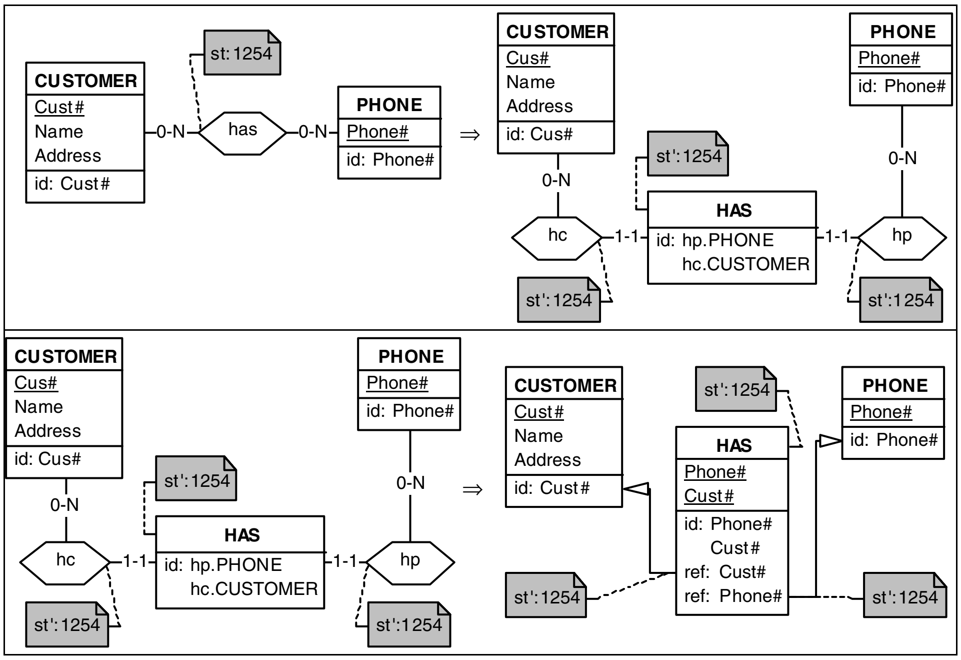


Fig. 9. Propagation of object stamps through two successive transformations.

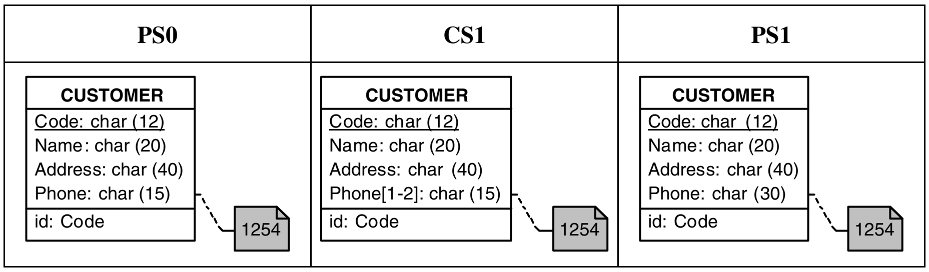


Fig. 10. Ambiguity in inter-schema mapping interpretation with the stamp technique.

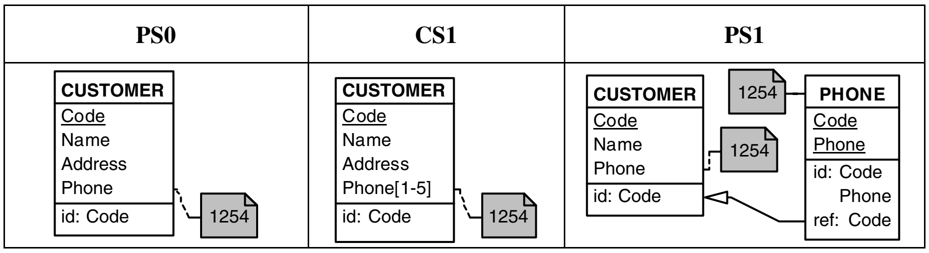


Fig. 11. Undefined complex transformation chains with the stamp technique.

To choose the right solution, the analyzer must know more about the conceptual modification, but it cannot decide only on the basis on the comparison between PS0 and PS1.

Second, a transformation chain can mislead the analyzer. In Fig. 11, the initially single-valued attribute *Phone* is changed to make it multivalued in CS1. Then, this multivalued attribute undergoes several complex transformations:

- The multivalued attribute *Phone* is instantiated into five single-valued attributes.
- Then, the last four of the resulting attributes are transformed into a multivalued attribute *Phone1*.
- Finally, the multivalued attribute *Phone1* is transformed into entity type *PHONE*.

Clearly, merely comparing the objects of PS0 and PS1 that share the stamp 1254 does not provide enough information to identify the correct transformation chain.

The synthetic approach requires a light evolution processor that is both easy to use and to maintain, since any new transformation/modification can be easily added to the comparison algorithm. The most frequent modifications can be processed with this technique. In many organizations, this approach is largely sufficient. The problematic cases like those described above appear to be infrequent and can be managed manually.

Representing and exploiting inter-schema mapping, though identified long ago, still is an active research domain. Schema mappings are used in many areas of database systems, including, of course, evolution, but also integration, migration, and view derivation and integration. Bernstein [5] has developed a framework, called model management, for managing schema mappings and their operators like composition. Data integration approaches have used schema mapping declarations in the form of global-and-local-as-view (GLAV) assertions [34].

4.6. Program modification

Modifying the application programs P0 according to database structure modifications is clearly a complex task that cannot be completely automated except in simple cases where the modifications are minor. To characterize the impact of data structure modifications on programs, we define three kinds of modifications:

- Some structure modifications do not require any modification to the programs. For example adding or modifying physical constructs (such as indexes and storage spaces) has no impact on the programs, at least for RDBMS.[3] The same is valid for the addition of a table or of columns for which no new constraints, such as *not null*, are defined.[4] Some more complex modifications can be absorbed through the *view* mechanism, if it can rebuild the former data structures.
- Other structure modifications only require minor, and easily automated, modifications to the programs. Such is the case with table renaming or value domain extension.
- However, many modifications involve deeper alteration of the program structure. They often require a thorough knowledge of the application. In Fig. 2/C, if the attribute *CUSTOMER.Account* becomes multivalued, it translates into a new table in the corresponding logical schema. Following this extension of schema semantics, the program must either keep its former specification (one must decide how to select the unique account value) or process the multiple accounts of each customer (generally by introducing a processing loop). In either case, the very meaning of the program must be managed, possibly through program understanding techniques. The difficulty lies in the determination of the code section that must belong to the new loop.

In most cases, the program modification is under the responsibility of the programmer. However, it is possible to prepare for this work by a code analysis that allows locating the critical sections of code. Program analysis techniques such as *pattern searching*, *dependency graphs,* and *program slicing* make it possible to locate with good precision the code most likely to be modified. These techniques have been detailed in [14,33].

That is on the basis of E, C1, and C0 analysis (Section 4.4), that the schema constructs that have changed can be identified and supplied to the program analyzers (dependency graph analyzers and program slicers). The latter locates the code sections depending on these modified constructs. A generator examines the results of the program analysis and produces a report of the modifications which would be advisable to apply to the programs under the programmer's control.

In the example of the attribute *PHONE*, the database conversion requires modifications of the program structure. Before the modification, the extraction of the telephone numbers of a person required a join operator, while in the new structure, the values are available in two distinct columns of the current *PERSON* row. Clearly, the processing of these values must be rewritten manually unless the program had been written in a particularly disciplined way. To locate the code to be modified, the program analyzers use parameters based on the table *PHONE*, its columns and the variables which depend on them.

Despite the intrinsic complexity of program evolution, new approaches are being developed, and may prove promising. One of them is based on wrapper technology[5] that isolates the application programs from the data management technology. A wrapper is used to simulate the source schema PS0 on top of the new database (schema PS1). The mappings *T* and *t* of the history E are encapsulated in the wrapper instead of being translated into program changes. In this way, the programs access the new data through the old schema, so that the application logic need not be changed. File I/O primitives have to be replaced with wrapper calls in a one-to-one way, a process that can be easily automated. This approach has been described and explored in [15]. Simpler approaches have been proposed, notably for object oriented applications. Ref. [21] for instance, describes a tool that finds the impacts of a schema change on the application components and shows them graphically. However, the lines of code that may be affected are not computed.

[3] This would not be true for legacy databases, such as standard files.

[4] Provided the *select* and *insert* statements use explicit column names.

[5] A *data wrapper* is a procedural component that transforms a database from one legacy model to another, generally more modern model. It appears as a data server to client applications and makes them independent of the legacy model technology. For instance a set of COBOL files can be dynamically transformed into an object store or into a relational database. Wrappers can automatically be generated by the transformational approach described in this paper. See [31] for more details.

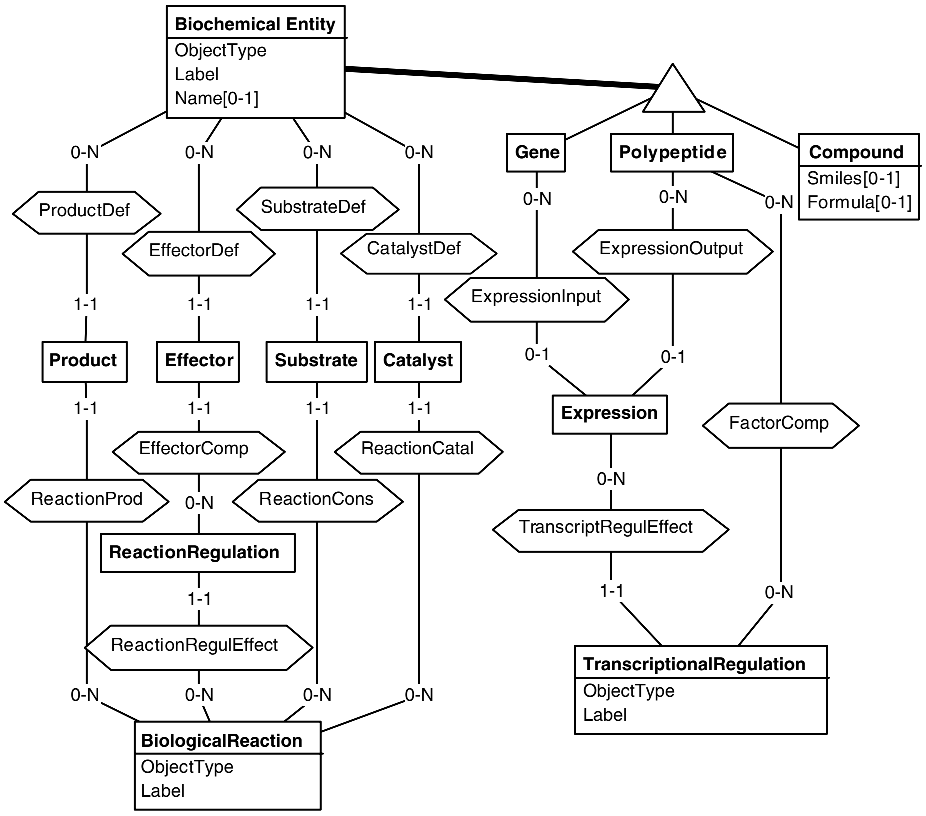


Fig. 12. Conceptual schema (CS0) of a subset of the Biomaze database.

5. Application: The Biomaze knowledge-based system

5.1. Presentation

The goal of the Biomaze project[6] is to develop a knowledge-based system for the representation, storage and dissemination of up-to-date knowledge on biochemical nets. The system, which is in construction, will include the following components.

- A biological database representing biochemical nets, implemented in the PostgreSQL relational DBMS.
- A tool-box of specialized processors to exploit biochemical networks: extraction, search, analysis, browsing, and graphical display.
- A web interface to access these processors.

Since the understanding of biochemical phenomena is evolving at a fast pace, the structure of the database must be changed quite frequently, which requires significant effort to modify the schema, convert the data, and adapt the processors accordingly.

Fig. 12 shows an excerpt of the conceptual schema of the Biomaze database (CS0). The full schema comprises 38 entity types and 41 relationship types, while the relational database that implements it is made up of 53 tables.

[6] The Biomaze project is developed in collaboration with the Catholic University of Louvain-la-Neuve (UCL, Department of Computing Science and Engineering) and the Free University of Brussels (ULB, Service of Biological Macromolecules Conformation and of Bioinformatics).

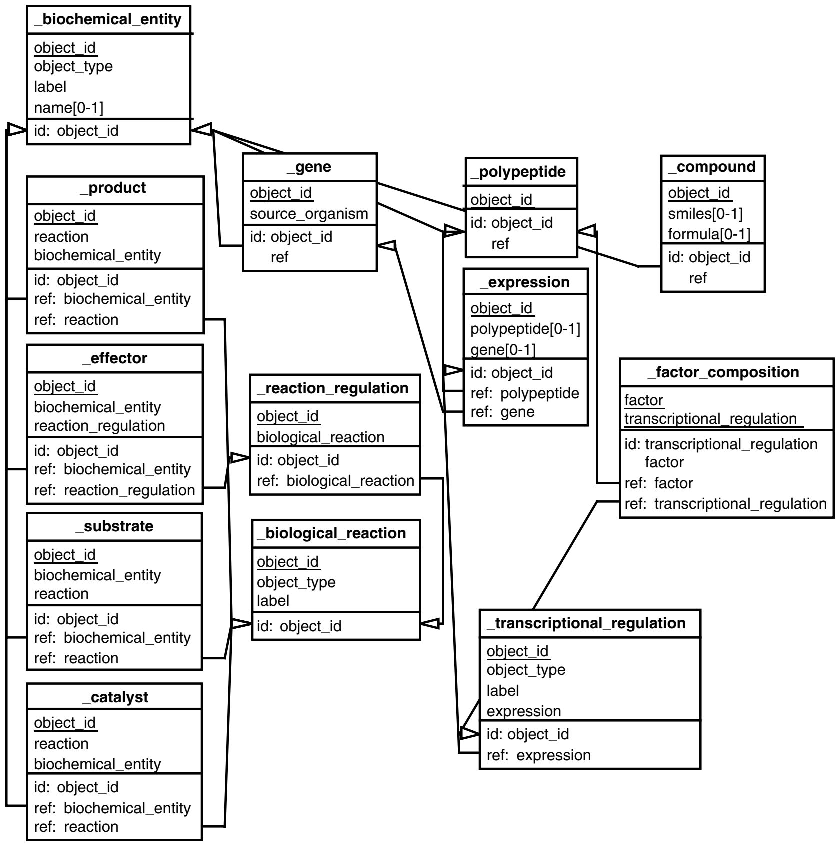


Fig. 13. Physical schema PS0 of a subset CS0 of the Biomaze database.

To simplify the discussion, and without loss of generality, we consider physical schemas in which indexes and storage areas have not been specified.[7] As a consequence, we can consider that the transformation of a conceptual schema into a physical schema is carried out in one step, through some kind of *logical–physical design* process, the execution trace of which is the history LPD0.

This history includes information on the operations necessary to transform the conceptual schema CS0 into the relational schema PS0. For instance, we will find there the description of the transformation of many-to-many rel-type *FactorComp* into an entity type, the addition of technical identifiers named *object_id*, transformations of all the rel-types into foreign keys, and modification of names (creating the indexes is ignored). A plausible version of PS0 is depicted in Fig. 13.

5.2. Conceptual modification

Due to environmental changes, the requirements currently met by the conceptual schema evolve. To illustrate the database evolution phenomenon, suppose we need two modifications of the conceptual schema CS0, leading to schema CS1, namely:

1. the *BiochemicalEntity.Name* attribute becomes multivalued,
2. the entity type *BiologicalReaction* is renamed into *BiologicalEvent.*

[7] Therefore, they are close to logical schemas. This is valid since indexes and storage areas imply no complexity in the process.

The conceptual evolution history (CE1) contains the following specifications:

```
CE1 = <
  T0: change-max-card(BiochemicalEntity.Name,N)
  T1: (BiologicalEvent) ← rename-ET(BiologicalReaction)
>
```

The schema fragments of Fig. 14 reflect these modifications.

5.3. The transformations in the Biomaze system

The analysis of the way biochemists make the conceptual schema evolve and of the conceptual-to-logical transformations that are currently used have lead to a fairly stable list of elementary transformations that is given in Fig. 15 for conceptual evolution and in Fig. 16 for relational logical design.

5.4. Schema modification propagation

The *logical–physical design* history that formerly transformed CS0 into PS0 is replayed on the modified conceptual schema SC1. The two modified structures are left unaltered in this process, so that the designer has to process them manually, or through an additional application of the *conceptual-to-logical-to-physical* transformation script, if the latter exists, which is usually the case when a CASE tool has been used.

The multivalued attribute *BiochemicalEntity* is transformed into entity type *_biochemical_entity_name* together with a one-to-many relationship type, which itself is transformed into foreign key *object_id* (Fig. 17, top right). The new *logical–physical design* history LPD1 includes the transformations of LPD0 augmented with those which were applied on the modified objects, i.e.,

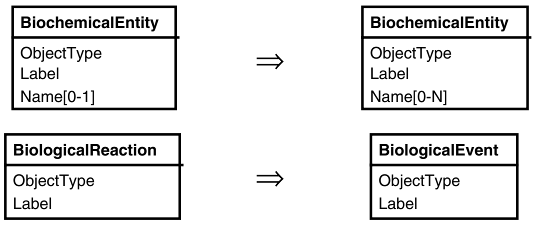


Fig. 14. Modification of the conceptual schema CS0 following requirements evolution.

	T-	T+	T=
Entity type	delete	create	rename ET-to-Att
Rel-type	delete	create	rename
Role	delete decrease max cardinality increase min cardinality	create increase max cardinality decrease min cardinality	rename
is-a relation	delete	create	
Attribute	delete decrease max cardinality increase min cardinality restrict value domain	create increase max cardinality decrease min cardinality extend value domain	rename Att-to-ET
Identifier	delete add component	create remove component	rename change status (prim./second.)

Fig. 15. The transformations for conceptual evolution that are monitored in the Biomaze system.

	T-	T+	T=
Entity type			rename split merge
Rel-type			RT-to-ET RT-to-FK
is-a relation			ISA-to-RT
Attribute			rename Att-to-ET disaggregate instantiate-multi multi-to-single

Fig. 16. The transformations for logical design that are monitored in the Biomaze system.[8]

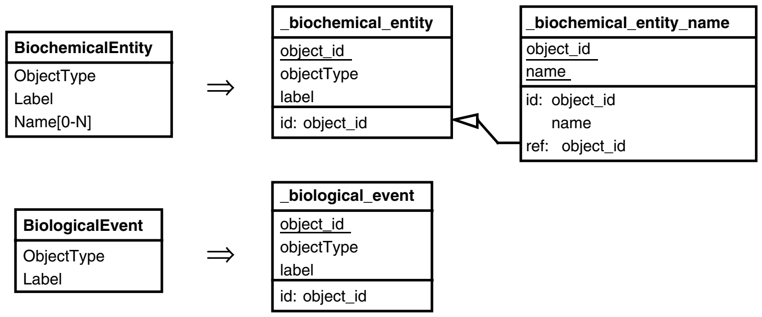


Fig. 17. Transformation of modified conceptual structures of CS1 into relational structures PS1.

```
LPD1 = <
... (transformations of LPD0)
T0: (_biochemical_entity_name,have) ← Att-to-ET-inst(_biochemical_entity.name)
T1: (_biochemical_entity_name.object_id) ← RT-to-FK(have)
>
```

5.5. Structure modification and data conversion

The database structures and instances must be converted according to the modification list. The converter generator analyzes the histories CE1, LPD0, and LPD1. A differential analysis gives the physical modification list between PS0 and PS1. In this case, this list contains the transformations T0 and T1 of LPD1, and T1 of CE1. After that, the converter translates these transformations into the following PostgreSQL script:

```
– Creation of table _biochemical_entity_name
CREATE TABLE _biochemical_entity_name (
  object_id NUMERIC(10) NOT NULL,
  name VARCHAR(1024) NOT NULL,
  CONSTRAINT id_biochemical_entity_name PRIMARY KEY (object_id, name));
```

[8] Transformation `instantiate-multi` has been illustrated in Fig. 8b. It replaces a multivalued attribute by a series of single-valued attributes. Transformation `multi-to-single` replaces a multivalued attribute by a single-valued attribute representing the concatenation of the values of the former.

```
– Transfert data
INSERT INTO _biochemical_entity_name
   SELECT object_id, name FROM _biochemical_entity
   WHERE name IS NOT NULL;

– Create foreign key
ALTER TABLE _biochemical_entity_name
   ADD CONSTRAINT fk_biochemical_entity
   FOREIGN KEY (object_id)
   REFERENCES _biochemical_entity;

– Create index
CREATE INDEX ind_biochemical_entity_name
   ON _biochemical_entity_name (object_id);

– Destruction of column name in table _biochemical_entity
ALTER TABLE _biochemical_entity DROP COLUMN name CASCADE;

– Rename table _biological_reaction into _biological_event
ALTER TABLE _biological_reaction RENAME TO _biological_event;
```

5.6. Program modification

In the Biomaze project, three approaches have been investigated for program modification:

1. On the basis of the modification list generated in the data conversion phase, the modified data structures are supplied to the program analyzer. In this case, the program slicing locates code sections depending on the table *_biological_reaction* and the column names *_biochemical_entity*. For the renamed table, a search and replace processor can realize the modifications. For the transformed column, a join operator is now necessary to read or write the new entity type *_biochemical_entity_name*. This modification is manual.

   Quite often, depending on the nature of the transformation, the use of relational views can minimize the change effort. In this case study, a view *_biological_reaction* is defined on table *_biological_event* by the instruction:

   ```
   CREATEVIEW_biological_reaction AS SELECT * FROM_biological_event;
   ```

   The programs remain unchanged. This solution can be applied when views can restore the data structures of CS0 from those of CS1. For the modification of attribute *name*, no view can be generated to simulate the old structure since the semantics itself has changed.
2. Wrapper technology can be chosen to isolate the programs from the data structures. Wrappers are automatically regenerated to simulate the schema PS0 on top of the new schema PS1. This technology is also used in progressive migration projects. The old program keeps running on the simulated old database, and can be migrated, at a convenient pace, to the new database [15].
3. A special type of wrapper, namely *parametric wrappers*, uses a dictionary of object names and access paths to dynamically generate the logical accesses to the database. A parametric wrapper is generic, in that it includes no direct references to schema objects. The program uses the services of the wrapper by sending it a query that includes the names of the objects of interest. The wrapper then consults the dictionary to check the validity of the query and to build the corresponding SQL query.

   Following the conceptual schema changes, a new version of the dictionary is generated. In this approach neither the programs nor the wrapper need to be modified. The Biomaze processors that run on the database are built following such an architecture.

6. Conclusions

The problem of the evolution of an information system, or data-centred application, includes that of the database which is at its core. Requirements evolution is translated technically into the modification of specifications at the corresponding abstraction level. The difficulty lies, on the one hand, in the propagation of these modifications towards the other levels and, on the other hand, to the operational components, namely the database and the programs.

The transformational paradigm appears to be particularly powerful in coping with these challenges. Indeed, modeling the engineering processes as transformations allows us to formally define and to reason on both the artifacts and the processes, whatever their granularity, ranging from elementary object modification to the whole process itself. This also provides a complete solution to the traceability issue.

The approach described in this paper addresses the common strategy in which evolution translates into the replacement of the source system with a new one, better suited to the current state of the requirements. Based on a generic data structure model and on a set of generic transformations, this approach can be customized to a large variety of data models and database management systems. It also seemlessly integrates the other engineering processes, such as forward engineering, reverse engineering, maintenance, migration, and federation.

The framework we have developed can be degraded into a lighter version of evolution control, the synthetic approach. This technique is much easier to implement and to support, and is more than adequate in most practical situations. In some infrequent cases, it may require manual intervention to solve ambiguities.

We have developed an extension of the DB-MAIN CASE tool [35] to support both the full and the synthetic approaches. This tool automatically generates the database converters corresponding to any transformation sequence. It also generates the rules to be used by the DB-MAIN program analyzers (dependency graph analyzer, program slicer) to identify the program code sections that should be modified.

An experiment has been carried out on a medium size database (326 tables and 4850 columns) in a distribution company. The application programs were made up of 180,000 COBOL lines of code distributed among 270 modules (a table appears in seven modules on average). The experiment showed that the time of assisted conversion of the structure, the data and the programs was less than one third of that of the manual process. With the assisted method, an engineer propagated one elementary database modification into the database and the programs in 1 day versus 3 days with the manual process. The company saved forty working days for 20 modifications a year while decreasing the risk of error. However, the assisted method required an initial investment of 30 days to rebuild a correct and up-to-date database documentation and to adapt the data conversion and program analysis tools.

The extended DB-MAIN tool is being integrated in Biomaze, a biochemical net knowledge base that includes a rapidly evolving database.

Research on large system evolution is far from closed, and this is particularly true for the present research. In particular, some important issues remain, such as extension of the DB-MAIN tool to the evolution of other non-relational data models, improved heuristics to make the synthetic approach more powerful in solving ambiguities, a better support for program migration, (notably based on the results presented in [15]), and the support of other evolution strategies, in particular those in which previous versions of the database still are available.

Acknowledgement

This work is partially supported by the Biomaze project, contract no. 315432, funded by the Région Wallonne.

References

[1] L. Al-Jadir, T. Estier, G. Falquet, M. Léonard, Evolution features of the F2 OODBMS, in: Proc. of 4th Int. Conf. on Database Systems for Advanced Applications, Singapore, 1995.
[2] J. Andany, M. Léonard, C. Palisser, Management of Schema Evolution in Databases, in: Proc. of 17th Int. Conf. on VLDB, Barcelona, 1991.

[3] C. Batini, S. Ceri, S.B. Navathe, Conceptual Database Design—An Entity–Relationship Approach, Benjamin/Cummings, 1992.
[4] Z. Bellahsene, An active meta-model for knowledge evolution in an object-oriented database, in: Proc. of CAiSE, Springer-Verlag, 1993.
[5] P. Bernstein, Applying model management to classical meta-data problems, in: Proc. og Conf. on Innovative Data Systems Research (CIDR), 2003, pp. 209–220.
[6] A. Brown, E. Morris, S. Tilley, Assessing the evolvability of a legacy system, Software Engineering Institute, Carnegie Mellon University, Technical Report, 1996.
[7] I. Comyn-Wattiau, J. Akoka, N. Lammari, A framework for database evolution management, in: G. Kniesel (Eds.), Proc. Workshop on Unanticipated Software Evolution (in conjunction with ETAPS'03), 2003.
[8] D. de Vries, J.F. Roddick, Facilitating database attribute domain evolution using mesodata, in: F. Grandi (Ed.), Proc. 3rd International Workshop on Evolution and Change in Data Management (ECDM2004), Shanghai, China, Springer, LNCS 3289, 2004.
[9] J.-L. Hainaut, M. Chandelon, C. Tonneau, M. Joris, Contribution to a theory of database reverse engineering, in: Proc. of WCRE, IEEE Computer Society Press, Baltimore, 1993.
[10] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland, Evolution of database applications: the DB-MAIN approach, in: Proc. of 13th Int. Conf. on ER Approach, Manchester, 1994.
[11] J.L. Hainaut, Specification preservation in schema transformations—Application to semantics and statistics, Data & Knowledge Engineering, Vol. 19, Elsevier, 1996, pp. 99–134.
[12] J.-L. Hainaut, J. Henrard, J.-M. Hick, D. Roland, V. Englebert, Database design recovery, in: Proc. of 8th CAiSE, 1996.
[13] J.-L. Hainaut, Chapter 1—Transformation-based database engineering, in transformation of knowledge, information and data: theory and applications, in: P. van Bommel (Ed.), IDEA Group, 2005.
[14] J. Henrard, V. Englebert, J.-M. Hick, D. Roland, J.-L. Hainaut, Program understanding in databases reverse engineering, in: Proc. of Int. Conf. DEXA, Vienna, 1998.
[15] J. Henrard, J.-M. Hick, Ph. Thiran, J.-L. Hainaut, Strategies for data reengineering, in: Proc. of WCRE'02, IEEE Computer Society Press, 2002.
[16] J.-M. Hick, Evolution of relational database applications: methods and tools, Ph.D. Thesis, University of Namur, 2001 (in French).
[17] J.-M. Hick, J.-L. Hainaut, Strategy for database application evolution: the DB-MAIN approach, in: Proc. ER'2003 Conference, Chicago, October 2003, LNCS Springer-Verlag, 2003.
[18] J.-H. Jahnke, J.P. Wadsack, Varlet: human-centered tool support for database reengineering, in: Proc. of Workshop on Software-Reengineering, 1999.
[19] M. Jarke, H.W. Nissen, K. Pohl, Tool integration in evolving information systems environments, in: Proc. of 3rd GI Workshop Information Systems and Artificial Intelligence: Administration and Processing of Complex Structures, Hamburg, 1994.
[20] C. Jensen et al., A consensus glossary of temporal database concepts, in: Proc. of Int. Workshop on an Infrastructure for Temporal Databases, Arlington, 1994.
[21] A. Karahasanovic, Identifying impacts of database schema changes on applications, in: Proc of the 8th Doctoral Consortium at the CAiSE*01. Freie Universität Berlin, Interlaken, Switzerland, 2001, pp. 93–104.
[22] X. Li, A survey of schema evolution in object-oriented databases, in: Proc. 31st International Conference on Technology of Object-Oriented Language and Systems. Nanjing, China, IEEE, 1999, pp. 362–371.
[23] P. McBrien, A. Poulovassilis, Schema evolution in heterogeneous database architectures, a schema transformation approach, in: Proc. CAiSE'02, 2002.
[24] T. Mens, J. Buckley, M. Zenger, A. Rashid, Towards a taxonomy of software evolution, in: Proc. 2nd International Workshop on Unanticipated Software Evolution, Warsaw, Poland, 2003.
[25] G.T. Nguyen, D. Rieu, Schema evolution in object-oriented database systems, Data & Knowledge Engineering, vol. 4, Elsevier Science Publishers, 1989.
[26] A. Rashid, Aspect-Oriented Schema Evolution in Object Databases: A Comparative Case Study. Lancaster, Computing Department, Lancaster University, UK, 2002.
[27] J.F. Roddick, N.G. Craske, T.J. Richards, A taxonomy for schema versioning based on the relational and entity relationship models, in: Proc. of 12th Int. Conf. on the ER Approach, Arlington, 1993.
[28] J.F. Roddick, A survey of schema versioning issues for database systems, Information and Software Technology 37 (7) (1995) 383–393.
[29] G. Shankaranarayanan, S. Ram, Research issues in database schema evolution—the road not taken, Working Paper #2003-15, University of Arizona, 2003.
[30] B. Shneiderman, G. Thomas, An architecture for automatic relational database system conversion, ACM Transactions on Database Systems 7 (2) (1982) 235–257.
[31] Ph Thiran, J.-L. Hainaut, Wrapper development for legacy data reuse, in: Proc. of WCRE, IEEE Computer Society Press, 2001.
[32] P. van Bommel, Database design modifications based on conceptual modelling, in: Information Modelling and Knowledge Bases V: Principles and Formal Techniques, Amsterdam, 1994, pp. 275–286.
[33] M. Weiser, Program slicing, IEEE TSE 10 (1984) 352–357.
[34] L. Xu, D. Embley, Combining the Best of Global-as-View and Local-as-View for Data Integration. ISTA 2004: 123–136.
[35] http://www.db-main.be/ and http://www.info.fundp.ac.be/libd.

Jean-Marc Hick, Master in Computer Sciences of the University of Namur (Belgium), has been working in the DB-MAIN research group since 1993. In 2001, he obtained a Ph.D. thesis with a dissertation on Evolution of relational databases applications: Methods and Tools. His research activities are concentrated on database evolution, data structure mapping, schema transformations, wrapper technologies and the DB-MAIN CASE tool. In December 2003, he participated to the creation of REVER, a spin-off of the LIBD (Laboratory of Database Applications Engineering). This company offers services and products in reverse engineering and reengineering of information systems.

Jean-Luc Hainaut is a full professor in Information System and Database Engineering at the University of Namur. He is head of the LIBD the purpose of which is to develop general methodologies and CASE tools for such engineering problems as database design, database reverse engineering, federated databases, database evolution, active databases, temporal databases, XML and web engineering. He is a co-author of the seminal paper of the Merise method, published in 1974. He is the author of several books on Database Modeling and Database Design and co-author of more than 80 journal and conference proceedings papers. He presented tutorials on conceptual modeling, transformation-based database engineering and database reverse engineering, notably in VLDB, ER and CAiSE conferences.