Database Evolution: the DB-MAIN Approach[1]

J-L. Hainaut, V. Englebert, J. Henrard, J-M. Hick, D. Roland
Institut d'Informatique, Univ. of Namur, rue Grandgagnage, 21 - B-5000 Namur (jlh@info.fundp.ac.be)

Abstract. The paper analyses some of the practical problems that arise when the requirements of an information system evolve, and when the database and its application programs are to be modified accordingly. It presents four important strategies to cope with this evolution, namely forward maintenance, backward maintenance, reverse engineering and anticipating design. A common, generic, framework that can support these strategies is described. It is based on a generic data structure model, on a transformational approach to database engineering, and on a design process model. The paper discusses how this framework allows formalizing these evolution strategies, and describes a generic CASE tool that supports database applications maintenance.

Key words. database evolution, database re-engineering, reverse engineering, transformational approach, process modeling, CASE tool

1. Introduction

1.1 The software engineering "failure"

Since the late seventies, models and methods have been developed to describe, analyse and design computer-based files and databases. CASE tools progressively emerged some years later. Currently, these models, methods and tools are widely spread and mastering them is considered as basics in software engineering culture and education. Entity-relationship (ER) and NIAM models, Database design methods, training programmes and CASE tools enacting them are now considered as standard and have been widely available from bookshops, software houses and vendors, from companies and from schools and universities for several years. In particular, dozens of graphical ER-based CASE tools are currently available from the market. One could conclude that data-intensive applications are now developed in a systematic way, and that they must be documented, well-structured, reliable and easy to maintain.

This, however, is known to be highly over-optimistic and far from reality. Models and methods are known in the field, but are seldom practiced, few companies buy CASE tools and a significant part of their owners do not use them on a large scale. This frustrating situation has lead some observers to describe it as the *software engineering failure.*

The reasons of the low penetration of the software engineering disciplines in application development are complex, and have been analysed for several years. One of the most important ones is the lack of support of critical phases of the software life-cycle, and particularly of database maintenance and evolution.

1.2 Incomplete database life-cycle and oversimplistic CASE technology

Considering first the apparent primary purpose of database design methodologies and CASE tools, that is the design of new database applications from scratch, it appears that they are based on a simplified model of the database life-cycle. Indeed, tasks that prove critical when designing medium to large-scale applications are ignored or at best weakly supported. For instance, cooperative design, conceptual views integration, full integrity support and user's views derivation are often lacking. In addition, important steps such as logical and physical design are strongly underestimated. As an illustration, the generation of SQL code from a conceptual schema is in most CASE tools fully automatic, and is based on simple rules only. Most often,

[1] The DB-MAIN project is partly supported by ARIANE-II (Be), Banque UCL (Lux), Centre de recherche public H. Tudor (Lux), CGER (Be), Cockrill-Sambre (Be), Computer Associates, CONCIS (Fr), DIGITAL, IBM, OBLOG Software (Port), Winterthur (Be), 3 Suisses (Be). It is developed in collaboration with the Database Laboratory of the EPFL (Lausanne, CH). The DB-PROCESS subproject is supported by the *Communauté Française de Belgique.*

this SQL code has to be considered as nothing more than prototype in all but small-scale applications. This SQL code is considered a useful product to validate the conceptual specifications, but inadequate as the operational database schema. The manual reworking of this schema through a text editor in order to give it decent performance is a common practice that is up to the designer, without any support from the CASE tool. Still worse, since the CASE tool is unaware of this schema restructuring, the mapping between the conceptual schema and the actual database structure is lost and cannot be used for further maintenance for instance. This practice inhibits the *traceability* property that the design should enjoy.

File and database life-cycles include other important activities once the application has been produced. Most current methodological products (methods and tools) completely ignore what happens *after birth*. Moreover, they ignore that most current developments do not concern new applications, but consist in extending and re-engineering existing applications. Methods and tools offer practically no support to essential processes such as migration, reverse engineering, re-engineering, conversion, evolution management, design replay, and the like.

1.3 Brief state of the art

The problem of database evolution has first been studied for standard data structures. Network and hierarchical schema restructuring has been addressed in [24], also recommended for earlier references. Relational (both 1NF and N1NF) schema modification is addressed in [1, 31, 35], among others. Suggested bibliography : [30]

The OO model has long been known as an elegant framework to solve the data structure evolution problems, particularly when processing aspects are concerned as well. Indeed, adding a component C to the type of object class O1 simply consists in deriving a subclass O2 whose type is that of O1 + component C. The methods that know about the existence/absence of C are redefined for O2. There is no agreement on how to cope with structure update propagation toward the object instances. However, this approach is of little help with traditional technologies, and particularly for legacy systems. See [26, 39, 30].

The impact of conceptual changes on relational schemas is studied in [31, 40]. However, only straightforward mapping rules are considered.

Several research projects have addressed the problem of change management at the level of the information system itself, including the processing components. Let us mention projects *DAIDA* [6, 19], *REDO* [34, 41], *MACS* [8] and *NATURE* [32, 20].

Schema consistency preservation is analysed in [9], while a taxonomy of changes for ER and relational structures can be found in [31].

1.4 The DB-MAIN project

DB-MAIN is a university/industry research project dedicated to the problems resulting from database evolution, including those related to poorly (or un-)documented applications. Its objective is to analyse these problems, and to discover the underlying concepts, techniques and strategies. It will put forward methodological guidelines and will develop specific CASE tools.

DB-MAIN inherits from two earlier projects dedicated to transformation-based database design (TRAMIS project) [15, 16] and to database reverse engineering (PHENIX project) [21, 17].

1.5 Contents of the paper

The paper is a wide overview of the DB-MAIN project. It is organized in four main parts.

The *first part* (section 2) presents the problem of database applications evolution, and discusses some important practical aspects presented as problem solving strategies. The *second part* (sections 3 to 5) describes the three paradigms on which the DB-MAIN proposals are built, i.e. the generic data structure model (section 3), the transformational approach to database engineering (section 4) and the design process modeling (section 5). The *third part* (section 6) is devoted to the DB-MAIN approach to database application maintenance. It shows how the three paradigms described in the second part can support the problem solving strategies The *fourth part* (section 7) describes the functionalities, architecture and components of a CASE tool that is aimed to assist the evolution of database applications.

2. Database Evolution - Problem Analysis

2.1 The problem

Making a database application evolve is a complex problem that induces different behaviours according to the complexity of the application, the quality of its design, and the culture of the development and maintenance personnel. This problem can be discussed by first considering the following abstract framework.

Let us consider a database that satisfies requirements `R0`. This database comprises schema `S0` and, at a given time, extension `E0`. Schema `S0` is made of the logical[2] schema `LS0` and the (optional) conceptual schema `CS0`. A set of database applications `P0` have been built for the database; they all work on the data through schema `LS0` (or external views of it).

Let us consider that requirements `R0` have changed into `R1`. In most cases, this change is translated into modifications of schema `S0` (`CS0`, `LS0` or both) leading to the new schema `S1` (`CS1` and/or `LS1`). If the conceptual schema `CS0` has been changed, the logical schema `LS0` must be changed accordingly (and conversely). Extension `E0` is no longer valid, and has to be converted into extension `E1`. Finally, the applications `P0` must be partly rewritten in order to comply with the new data structures described in `LS1`.

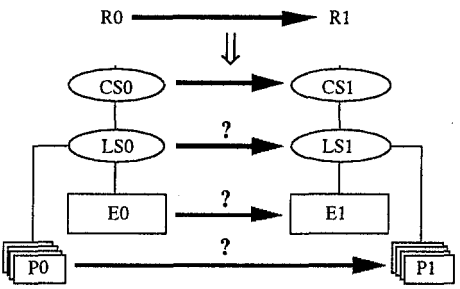

Fig. 2.1 - Intuitive representation of the database evolution problem as modeled in the DB-MAIN approach. A change in requirements R0 is translated into a change in a schema (e.g. CS0). The problem is to propagate this change into the logical schema (LS0), the data (E0) and the programs (P0).

This description allows us to pinpoint the major practical problems that will occur, and that are addressed by the DB-MAIN project. With each of these problems, a typical maintenance strategy will be associated.

2.2 Strategy 1 : Forward database maintenance

The first problem is described in a (somewhat idealistic) context in which both `CS0` and `LS0` schemas are available and correct. Let us suppose that the change in requirements `R0` have been translated into a change in the conceptual schema `CS0` in such a way that `CS1` now satisfies `R1` (e.g. adding an attribute, changing the cardinality of a role, splitting an entity type). The basic problem is *how can this change be (as far as possible automatically) propagated down to the logical schema `LS0`, data `E0` and application programs `P0` to produce `LS1`, `E1` and `P1` ?* It can be called the *forward database maintenance* problem.

A schema modification occurs as the consequence of a change in the requirements of (the database component of) the information system. First, the relevant schema must be retrieved. If the requirements describe the user expectations (*functional requirements*), their change will be converted into a change in the conceptual schema. On the contrary, if they specify

---

[2] In this description, the logical schema denotes the specifications of the data structures that are provided to, say, the application programmer. This schema complies with the model of the DBMS, and will sometimes be designated as the DBMS schema. For simplicity, the physical schema and user's views are ignored.

technical constraints or goals of the final system (*non functional requirements*), they will concern lower-level schemas, in which criteria such as performance and security are addressed.

Automatic or assisted update propagation requires the knowledge of the exact mapping between the conceptual and logical (or DBMS) schemas[3]. Indeed, this mapping allows to replay[4] most processes that were carried out when building the old system. The designer is only responsible for controlling the propagation of the new constructs. When the new schema has been produced, it should be as close as possible to the old one, except for the structures that have been concerned by the changes. Following this schema updating, the database contents must be converted as well. According to the kind of change, some data will be discarded, some will be converted, while some data structures will be left empty. Finally, the application programs will be updated in order to let them operate on the new database. In the current state of the art, it is impossible to completely solve this conversion problem, but much help can be brought to the programmer in this task.

The way this DBMS schema has been derived from the conceptual one will determine the complexity of this mapping, and its availability. In most CASE tools, the DBMS schema is automatically generated from the conceptual schema, with no (or little) action from the designer, through a limited set of simple translation rules. Therefore, regenerating a new DBMS schema after modification of the conceptual specifications is a straightforward process.

Unfortunately, automatic production of DBMS schemas by current CASE tools has proved unsatisfactory as far as *non functional requirements* are concerned. This situation is due to the over-simplistic translation rules, to the absence of important information regarding, e.g., statistics and data use, and to the weakness (or absence in most cases) of expertise concerning database optimization included in these tools. Implementing the database from the unchanged generated schema generally leads to poor performance. Therefore, the generated DBMS schema will most often be restructured manually by the designer[5]. Since this restructuring is done outside the tool, the conceptual/DBMS mapping is lost. As a consequence, these tools are practically useless in database maintenance, even for conceptual/DBMS schema update propagation alone.

2.3 Strategy 2 : Backward database maintenance

Let us suppose that the conceptual schema `CS0` is still available and correct. Very frequently, due to time constraints and current CASE tools weaknesses, the change in requirements `R0` are directly translated into changes in the logical schema `LS0` in such a way that `LS1` now satisfies `R1` (e.g. adding a column to a table). The second problem is *how can these changes in `LS0` be propagated up to the conceptual schema `CS0`, giving `CS1` that reflects the semantics of the new logical schema `LS1` ?* This process can be called *backward database maintenance.*

Backward maintenance is certainly not the cleanest way to support database evolution. However, it corresponds to common practice. Moreover, when compared to forward maintenance, it could be easier to implement in a context where traditional CASE tools are used. Supporting this practice is perceived by some practitioners as a first step in database maintenance.

Though it seems linked to database reverse engineering, this problem is significantly different in two aspects. On the one hand, the conceptual schema of the old version of the database is available and is correct. On the other hand, most variations between old and new DBMS versions are minor : modify a field length, add, remove or rename a field, add or drop a file.

---

3 The forward mapping specifies how each conceptual object has been implemented in the database. The backward mapping specifies, for each data structure of the database, the conceptual construct(s) it is an implementation of.

4 The term of *replay* is used in software engineering to designate the reexecution of design activities that have already been carried out to produce the former version of a system.

5 A worse practice consists in restructuring the conceptual schema in order to make it satisfy the technical requirements. Such a schema can no longer be considered as conceptual. Indeed, this restructuring would depend on the target DBMS, and would not be *computer-independent.* For instance, starting from a pure conceptual schema, there would be SYBASE-oriented or DB2-oriented conceptual schemas !

Here too, the feasibility of systematically updating the conceptual schema depends on the availability and of the complexity of the conceptual/DBMS mapping.

Knowing how each conceptual construct has been translated into DBMS data structures (forward mapping), one can deduce the backward mapping, i.e. from what conceptual construct(s) a given DBMS construct was derived. Modifying the DBMS schema must trigger a modification of the old conceptual schema in such a way that the new conceptual schema would lead to the new DBMS schema, *should the same translation rules as earlier be applied.*

Since most current CASE tools do not maintain the mapping from the conceptual schema to the *actual* DBMS schema (i.e. possibly reworked according to performance requirements for instance), they cannot support backward maintenance.

Note that data conversion (as well as program conversion) can be tackled through forward maintenance. However, the developer has probably already carried out this conversion manually.

2.4 Strategy 3 : Database reverse engineering

In many situations, particularly when so-called *legacy systems* are considered, the only description available is the source code of the file and database description and of the manipulation programs. Schema `CS0`, and sometimes (for standard file applications[6] for instance) schema `LS0` are missing. The third problem is as follows : *how can schemas `LS0` and `CS0` be recovered in order to make the system evolve securely ?* This process is called *database reverse engineering.* It allows applying strategy 1 (or 2) on old and poorly documented systems.

Reverse engineering pieces of software has often been discussed in the context of system maintenance. For instance, [41] (project REDO) and [8] (MACS environment) propose it as a way to support the evolutive maintenance process. Database reverse engineering can be conducted in two phases [34, 17].

The first phase, called *Data Structure Extraction*, consists in recovering the DBMS schema. True database systems generally supply, in some readable and processable form, a description of this schema (data dictionary contents, DDL texts, etc). Though essential information may be missing from this schema, the latter is a rich starting point that can be refined through further analysis of the other components of the application (views, subschemas, screen and report layouts, procedures, fragments of documentation, etc). The problem is much more complex for standard files, for which no computerized description of their structure exists. Source programs must be analysed in order to detect partial structures of the files. This analysis goes well beyond the mere detection of record structures declared in the programs. For instance, field decomposition or important integrity constraints can only be detected by a careful analysis of the procedural statements and of the local variables. These partial structures must then be refined and merged to obtain the unique DBMS schema.

The second phase, called *Data Structure Conceptualization*, is the conceptual interpretation of the DBMS schema. It consists for instance in detecting and transforming or discarding non-conceptual structures, redundancies, technical optimization and DBMS-dependent constructs [18]. Several proposals have addressed this phase, generally for specific DBMS, and for rather simple schemas (e.g. with no implementation tricks). Let us mention, in addition to the references mentioned above, [27, 7, 25, 37, 43, 3, 11, 29].

During these processes, the mapping between the schemas is recorded, in order to be further processed for purposes such as documentation, data administration, database evolution (through forward maintenance for instance), database conversion, etc. In particular, a redocumented database can be maintained according to the procedures that have been discussed hereabove.

As the main output of the PHENIX project [14, 21, 17, 18], a generic, DBMS-independent, methodology has been developed and implemented into CASE tools, and is a major baseline of the DB-MAIN approach.

---

6 As discussed in [17], there is no global schema of the files used by a COBOL application. There are as many partial views of these files as there are programs using them.

2.5 Strategy 4 : Anticipating design

It is widely accepted that some design and programming styles lead to better tolerance to requirements evolution than others. Hence the fourth problem : *what reasonings and design techniques can be proposed to designers and programmers to build database structures (CSO and LSO) and applications (PO) that are more robust against changes of requirements ?* This strategy will not be addressed in this paper.

2.6 Requirements for a maintenance environment

These strategies are deeply intricated with the processes and products that appear in traditional database design. Therefore, a database application maintenance methodology (and tools) must be an extension of a database design methodology[7] (and tools).

In short, an environment that is to assist in applying these strategies, their combinations and variants, must satisfy some requirements that include the following.

1. *The environment must completely represent data structures at any level of abstraction.* Indeed, the analyst will have to manipulate, and to reason about conceptual, logical and physical schemas. This description must be complete, otherwise some specifications will escape the control of the environment, and reasoning on them will be impossible.
2. *The environment must be DBMS-independent.* It must be able to represent precisely data structures according to any operational data management systems, be they standard file systems or advanced RDBMS[8].
3. *The relationships between data structure descriptions must be completely formalized.* Complex applications require a large number of data descriptions. The mappings between these descriptions must be analysed and manipulated in order to reason on them and to derive new information. Two of the most obvious mappings are the forward mapping (stating how each conceptual object has been translated into physical objects) and the backward mapping (stating what conceptual objects each physical object is an implementation of). Mere informal relationships such as "schema Si *derives from* schema Sj" cannot be a sound basis for that analysis and manipulation.
4. *The environment must completely represent, and therefore support, all database engineering activities.* All the operations that are carried out on the data structure specifications must be described precisely. The completeness requirement is essential. Indeed, if a manipulation (e.g. schema optimisation) cannot be described by the environment, the latter can no longer control this manipulation.
5. *The environment must be methodology-independent.* At the present time, there is no strong agreement on standard design methodologies. In addition, many files and databases have been (and still are) designed through empirical approaches and some design activities are not formalized yet, e.g. database reverse engineering. The environment must be flexible enough to describe precisely all these design behaviours.

These requirements imply two important properties of the approach : **genericity** and **formality**. Genericity must allow for, a.o., DBMS-independence, methodology-independence, reuse of reasonings and functions. Formality will bring precision and security of the description and the reasonings as well as full control of the environment on the specifications and on their manipulation.

The following three sections will present briefly the basic paradigms on which the DB-MAIN approach is built, namely

- a unique generic model to represent all specifications states during the database life-cycle,

---

[7] Another example : conceptual modeling often implies reverse engineering aspects, while reverse engineering a database include conceptual normalization of the resulting schema.

[8] OO-DBMS are not discarded, but the most urgent and complex problems so far concern applications that use traditional data management systems.

- the transformational approach of database engineering,
- database design processes modeling.

3. Unique, Generic, Data Structure Model

Database engineering is concerned with building, converting and transforming database schemas at different levels of abstraction. Elaborating DMS-independent and even methodology-independent techniques and reasonings that support these activities requires the availability of a set of models to express all these schemas. Due to the transformational approach adopted in this project (section 4), and due to the large scope of the proposal that encompasses all the traditional levels of abstraction, it has been found essential to base it on a unique, generic, schema specification model[9]. This model and its transformational operators are intended, (1) to support forward as well as reverse engineering, (2) to express conceptual, logical and physical schemas, as well as their manipulation, (3) to support any DBMS model and the production and manipulation of its schemas. Its ability to represent in a unique formalism such a variety of schemas makes this model an ideal framework to express inter-schema mappings, an essential requirement in database maintenance. The model includes three layers.

The **organizational layer** describes the higher-level structure of the specifications. All the specifications are collected into a project. A project is made of schemas (data structures), processing descriptions (ignored in this presentation) and relationships between them all.

The **conceptual layer** of the model, limited to data structure specifications, consists of the features of the standard ER model with some extensions : entity type and *is-a* hierarchy; relationship type (*rel-type* for short) with single/multi-ET roles and cardinality constraint [min-max]; atomic/compound attributes, single/multi-valued attributes (with cardinality [min-max]), pure/bag/list multivalued attributes; entity type, rel-type and attribute identifiers, made of attributes and/or roles; various integrity constraints (inclusion, referential, redundancy, exclusion, coexistence, functional). Some of these constructs are illustrated in figure 3.1.

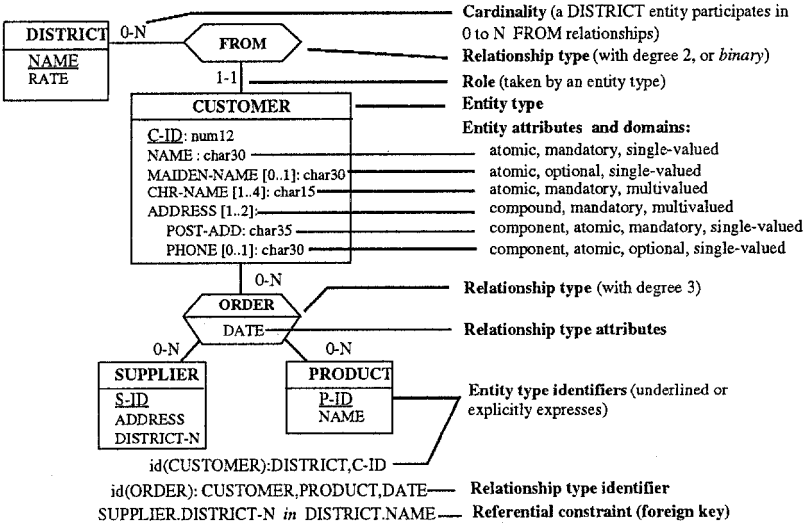

Fig. 3.1 - Graphical representation of some constructs of the generic model.

---

9 According to [6] that follows F. L. Bauer, this model can be called **wide-spectrum** since "it offers constructs for specification ranging from highly abstract [...] to ones that are implementation oriented".

The **technical layer** includes constructs that pertain to the description of logical and physical data structures : files, access keys, access paths, sequences, physical length and position, overlapping attributes.

This model must be considered as a generic specification model that can be specialized into a great variety of submodels, for instance according to the design levels or design product classes of a standard or user-defined multilevel design method, or according to the target DBMS (section 5). A more comprehensive description of the model and of its specialization techniques can be found in [15], while its formalization has been proposed in [12].

4. Transformation-Based Database Engineering

In general, a schema transformation consists in deriving a target schema S' from source schema S by some kind of local or global modification. Adding an attribute to an entity type, deleting a relationship type, and replacing a relationship type by an equivalent entity type, are three examples of schema transformations. The proposed approach states that producing a database schema from another schema can be carried out through selected transformations. For instance, normalizing a schema, optimizing a schema, producing an SQL database or COBOL files, or reverse engineering standard files and CODASYL databases can be described mostly as sequences of schema transformations. Some authors propose schema transformations for selected design activities [3, 22, 24, 33][10]. Moreover, some authors claim that the whole database design process, together with other related activities, can be described as a chain of schema transformations [4, 18][11]. Schema transformations are essential to formally define forward et backward mappings between schemas, and particularly betwen conceptual structures and DBMS constructs. The notion has been defined in [13], and can be summarized as follows.

4.1 Principles

A **transformation** Σ consists of two mappings **T** and **t** :

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

According to the context, Σ will be noted either **<T,t>** or **<P,Q,t>**. The implication of these definitions in the problem of database maintenance is rather straighforward. Let us consider, for example, that transformation Σ has been applied on logical schema `LS0` due to a change in the requirements (entity type splitting to satisfy disk space requirements for instance). We can easily state[12] that :

- mapping **T** indicates how the new schema `LS1` can be obtained,
- mapping **t** specifies how to convert extension `E0` into new extension `E1`,
- if `J` is a data type that has been affected by mapping **T**, each program statement operating on object `j` of type `J` in database extension `E0` must be changed in such a way that it now works on object **t**(`j`) of type T(`J`).

---

10 A more comprehensive bibliography can be found in [18].

11 It is interesting to note that this approach has long be considered in pure software development. According to [2] and [10] for instance, *the process of developing a program [can be] formalized as a set of transformations.*

12 This is only an very intuitive view. Things are a bit more complex to formalize, specially in multilevel designs and when many transformations have been applied.

4.2 Reversibility

Some transformations augment the semantics of the source schema (e.g. add an attribute), some remove semantics (e.g. remove an entity type), while others leave the semantics unchanged (e.g. replacing a relationship type with an entity type). The latter are called *reversible* or *semantics-preserving*. A transformation is reversible if the source and the target schemas have the same descriptive power, and describe the same universe of discourse.

- A transformation $\Sigma 1$ = <P1, Q1, t1> = <T1, t1> is reversible, *iff* there exists a transformation $\Sigma 2$ = <P2, Q2, t2> = <T2, t2> such that, for any construct C, and any instance c of C: P1(c) ⇒ ([T2(T1(C))=C] and [t2(t1(c)=c]). $\Sigma 2$ is the inverse of $\Sigma 1$, but, surprisingly, not conversely. For instance, an arbitrary instance c' of T(C) may not satisfy the property c'=t1(t2(c')).
- If $\Sigma 2$ is reversible as well, then $\Sigma 1$ and $\Sigma 2$ are called symmetrically reversible. In this case, $\Sigma 2$ = <Q1, P1, t2>, and both transformations can be defined through the unique notation Σ = <P, Q, t1, t2>. It is called an SR-transformation. This is the most desirable form, and most techniques that are proposed in the DB-MAIN approach have this SR-property[13]. However, in database design, and particularly at the implementation level, non fully reversible transformations may be used due to the unavailability of SR-transformations.

Similarly, in the pure software engineering domain, [2] proposes the concept of *correctness-preserving* transformation aimed to compilable and efficient program production.

4.3 Notation

Though these definitions do not depend on any specific data structure model, we will concentrate on ER schemas. A PROLOG-like specification of T through the expression of its components P and Q is suggested in [15]. However, we will use a more readable expression in which generic versions of C and T(C) are represented through ER graphical convention. As an illustration, fig. 4.1 describes how a *many-to-many* rel-type (i.e. C) can be replaced by a new entity type, by two *one-to-many* rel-types, and by an identifier (i.e. T(C)).

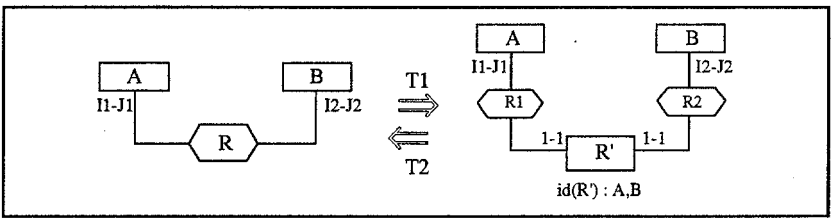

Fig. 4.1 - Representation of structural mapping T1 (from left to right) & T2 (from right to left) of a typical SR-transformation.

t1 : for each r(a,b) in R do generate arbitrary entity r' in R' insert (r',a) in R1 insert (r',b) in R2	**t2** : R = R1 ∘ R2

Fig. 4.2 - Representation of instance mapping t1 (from left to right) & t2 (from right to left) of a typical SR-transformation. Mapping t2 is expressed as the composition of relationship types R1 and R2.

---

13 [3] gives another definition of reversibility in terms of information equivalence of source and target schemas : "schemas S1 and S2 have the same information content (or are *equivalent*) if for each query Q that can be expressed on S1, there is a query Q' that can be expressed on S2 giving the same answer, and vice versa." It is easy to prove that SR-transformations are mappings between schemas that have the same information content according to this definition.

This transformation is generic since names A,R,R1,R2,I1,J1, I2,J2 must be replaced by actual values (e.g. CUSTOMER, PLACES, . . , 1 , 1) in order to get a specific transformation on an actual schema.

The t part of these transformations can be expressed as in fig. 4.2.

4.4 Some examples

We will give two other examples for which only the graphical expression of T will be given. They are borrowed from [13, 18] in which other standard transformations can be found.

One of the most used transformations replaces a one-to-many relationship type by a *foreign key* (Fig. 4.3), which can be multivalued (cardinality J > 1) or single-valued (cardinality J = 1).

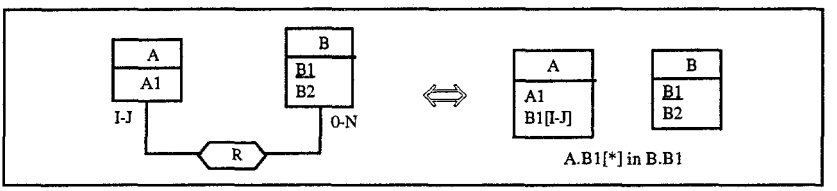

Fig. 4.3 - Rel-type R is represented by foreign key B1.

Transforming an attribute into an entity type is another standard common technique. It comes in two flavours, namely **instance representation** (Fig. 4.4(a)), in which each instance of A2 in each E entity is represented by an EA2 entity, and **value representation** (Fig. 4.4(b)), in which each distinct value of A2, whatever the number of its instances, is represented by an EA2 entity.

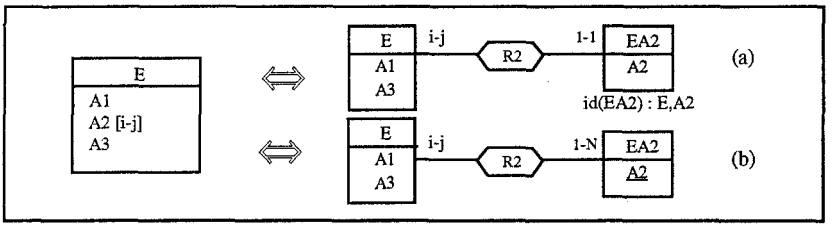

Fig. 4.4 - Transformation of an attribute into an entity type : (a) by explicit representation of its instances, (b) by explicit representation of its distinct values.

5. Design Process Modeling

Database evolution is strongly linked with the way the current database and its applications were developed, and the requirements they were to satisfy, i.e. problems that are addressed by software process modeling. (Software) *process modeling* is concerned with the understanding, representation and computer-based support of the software engineering activities [28]. The latter can be modeled as a set of products (schemas, programs, reports, etc) and a set of design processes that transform input products into output products according to specific requirements to satisfy. For instance, in the database realm, conceptual user's views are integrated into a unique conceptual schema by an integration process, the conceptual schema is normalized into a canonical conceptual schema through a normalization process, this schema is translated into an SQL schema by a DBMS-oriented translation process, the latter schema is then optimized, and finally coded by other ad hoc processes. Each process in turn can be described as a set of products and processes, until primitive processes can be described. The model we have chosen derives from proposals such as [28, 32]. One of its characteristics is that the requirements are distributed among the design processes that form a *methodology*. It describes quite precisely not only standard methodologies, such as the Conceptual-Logical-Physical approaches [3, 38] but also any kind of heuristic design behaviours, including those that occur in reverse engineering. Here follows the definition of some of these concepts.

5.1 Product and Product instance

A product instance is any outstanding specifications object that can be identified in the course of a specific design. A conceptual schema, an SQL DDL text, a COBOL program, an entity type, a table, a collection of user's views, an evaluation report, can all be considered product instances. Similar product instances are classified into products, such as CONCEPTUAL_SCHEMA, NORMALIZED_ BINARY_SCHEMA, SQL_DDL_SCHEMA.

There is a limited set of *generic products*, such as TEXT (any textual material) and GER_MODEL (the generic ER model described in section 3). Any other product type P2 can be derived from another generic or derived product type P1 by four specialization techniques :

- subsetting : P2 includes a subset of the concepts of P1 (e.g. no IS-A relations);
- constraints : the constructs allowed in P2 are more constrained than those in P1 (e.g. binary relationship types only);
- enrichment : P2 can include constructs that are ignored in P1 (e.g. access specifications);
- renaming : the same concept is given different names (the concept of ATTRIBUTE in P1 is renamed COLUMN in P2).

5.2 Process and Process instance

A process instance is any consistent activity in a design which transforms[14] a product instance into another product instance. Normalizing schema S1 into schema S2 is a process instance. Similar process instances are classified into processes. NORMALIZATION is a process. There are two categories of processes, namely design processes and primitives.

- A *design process* is a goal-oriented process that is intended to make its input product satisfy specific functional or non-functional[15] requirements. NORMALIZATION, TIME_ OPTIMIZATION and REVERSE_ ENGINEERING are examples of design processes.
- On the contrary, a process is a *primitive* if it is a deterministic atomic operation; generally, a primitive is neutral w.r.t. the requirements (it has no goal). Another difference is that the strategy of a primitive is encapsulated and is carried out by the CASE tool, while the strategy of a design process is visible, and has to be carried out by the designer, or at least under its control. The creation of an entity type and the transformation of a attribute into an entity type are examples of primitives.

5.3 Process strategy

The strategy of a process is the specification of how its goal can be achieved, i.e. how the process can be carried out. A strategy can be deterministic, in which case it reduces to an algorithm (and can often be implemented as a primitive), or it can be non-deterministic, in which case the exact way in which each of its instances will be carried out is up to the designer. The strategy of a design process is defined by a *script* that specifies, a.o., what lower-level processes must/can be triggered, in which order, and under which condition.

The control structures in a script include *action selection* (at most one, one only, at least one, all in any order, all in this order, at least one any number of times, etc.), *alternate actions*, *iteration*, *parallel actions*, *weak condition* (recommended satisfaction), *strong condition* (mandatory satisfaction), etc.

---

14 This model is a generalization of the transformational approach developed in section 3. In particular, the concept of reversibility can be applied on processes, which are higher-level schema transformations.

15 This classification has been developed in the DAIDA project [6] : ***functional requirements*** describe what the information system under development is intended to do (its functionality), how it interact with its environment, what information it will manage, and how that information relates to the system's environment. The ***non-functional requirements***, or goals, describe desirable properties of the intended system, they impose global constraints on the operation, performance, accuracy and security of any proposed solution to the functional requirements.

5.4 Hypothesis, decision and rationale

In many cases, the designer will carry out an instance of a process with some hypothesis in mind. This hypothesis is an essential characteristics of this process instance since it implies the way in which its strategy will be performed. When the designer needs to try another hypothesis, (s)he can perform another instance of the same process, generating a new instance of the same product. After a while (s)he is facing a collection of instances of this product, from which (s)he want to choose the best one (according to the requirements that have to be satisfied). A justification of the decision must be provided. Hypothesis and decision justification comprise the design rationale [23]. The decisional aspects of process modeling are particularly addressed in [28] (through the concept of *deliberation*) and in NATURE [32].

5.5 Product version

When more than one instance of the same product is generated, each of them is called a version. There are two kinds of versions.

- First, as discussed hereabove, a collection of versions can be instances of the same object that derive from a series of distinct hypotheses. All these versions are supposed to satisfy the same requirements, but possibly at different levels. Before going on, one of these versions (the best one) has to be chosen, while the others are discarded.
- Secondly, a collection of versions can be instances of the same object that satisfy different requirements. Each version can be brought to its final term (e.g. an SQL DDL text). For instance, each version is dedicated to a different H/S platform. Making a software product (e.g. a database) evolve consists in building a new version.

In some approaches (deriving from the Nilsson work), these categories define respectively OR / AND nodes in the hierarchy of the product versions (or process instances) of a history [36].

5.6 History of a process instance, of a project, of a product instance

The history of a process instance is the recorded trace of the way in which its strategy has been carried out, together with the product instances involved and the rationale that has been formulated.

Since a project is an instance of the highest level process, its history collects all the design activities, all the product instances and all the rationales that appeared, and will appear, in the life of the project.

The history (also called the *design*) of product instance P is the set of all the process instances, product instances and rationales which contributed to P. The design of a database collects all the information needed to describe and explain how the database came to be what it is.

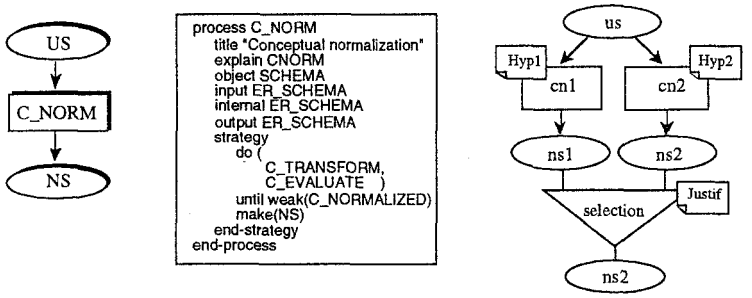

Fig. 5.1 - Graphical representation of some aspects of the process model. Left-hand : a process C_NORM is defined as a transformation of product US (un-normalized schema) into NS (normalized schema). Middle : this text specifies this process, a.o. by declaring its strategy as a script (carry out transformations and evaluations until the current schema satisfies - preferably [weak] - the submodel C_NORMALIZED). Right-hand : a history of C_NORM that shows two instances cn1 and cn2, carried out on product instance us, and having produced instances, or versions, ns1 and ns2 of NS. Hypotheses have been associated with each of these process instances. One of these product instances is selected, and this decision is justified.

6. The DB-MAIN Approach

DB-MAIN has started in 1993, and it is too early to propose comprehensive strategies that can solve the various aspects of the maintenance problem. So, we will only sketch the way these problems can be coped with on the basis of the principles developed in the sections above.

6.1 Strategy 1 : Forward database maintenance

This strategy makes the hypothesis that the reference methodology, as well as the history describing formally all the aspects of the database design are available. If it is not the case, the situation first falls in the reverse engineering domain. We will discuss the modification of the three major components, namely the schemas, the data and the programs.

Step 1 - Changing the database schemas

The techniques for changing the conceptual schema, then propagating these changes to the DBMS schema are based on the following reasonings.

- Identifying the changes in requirements `R` allows to locate the process[16], say `P`, that is dedicated to `R`. Let us call `p` the relevant instance of `P` in the history, `si` and `so` their input and output product instances (they typically are schemas).
- All the schemas and process instances that precede `p` in the history, including `si`, still are valid, and can be kept unchanged.
- `P` must be performed again on `si`, leading to `p'`, a new instance of `P` and `so'` the new version of the output schema. During `p'`, the designer incorporates the requirements change. In general, the reasonings applied in `p` can also be applied in `p'`, except for some minor aspects related to the change. So, the designer can merely **replay** the actions of `p` that are not involved in the change, concentrating only on those which concern the change.
- Once the new schema `so'` is available, the designer has to replay all the processes instances for which `so` was a direct or indirect input schema. The history can tell what actions can be replayed automatically, and what actions must be carried out with help from the designer when they concern modified constructs.
- Finally, new DBMS schema `sf'` replaces old schema `sf`.

These concepts and their evolution are sketched in fig. 6.1.

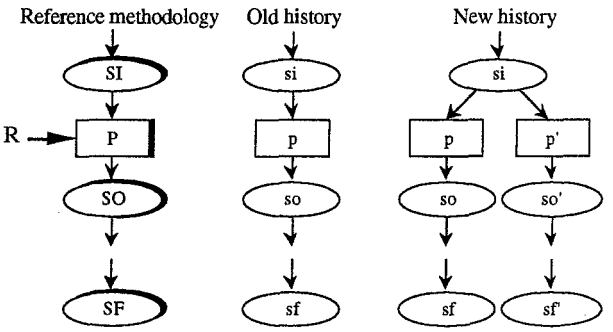

Fig. 6.1 - Translating a requirements change into schema modification. The relevant process `P` is identified, and its instance `p` retrieved. A new version `p'` of `p` is carried out by replaying of the latter, during which the needed change is carried out on the schema. Then all the subsequent processes of the old history are replayed until a new final schema `SF'` is obtained.

A taxonomy of ER schema modifications is proposed in [31]. For each of them, it suggests a propagation rule that impacts this change in the relational translation of the ER schema. This approach is based on straighforward ER/Relational mapping rules that do not address more

[16] There can be more than one. We will ignore this fact in this presentation.

complex design reasonings that may occur when non functional requirements (e.g. performance) are processed.

Step 2 - Changing the database contents

Let us first suppose that process P is terminal, i.e. that sf = so and sf' = so'. If the incorporation of the change can be expressed as transformation Σ = <T,t> (which is the basic hypothesis of the approach) acting on final schema so, then its **t** part states what data transformation should apply. Practically, this principle leads to several cases of change M, among which we can mention the following.

- The initial schema change M (which occurred during p') does not alter the structure of the schema (e.g. renaming some objects) : no data change needed.
- M consists in relaxing (or generalizing) a constraint : no data change needed.
- M consists in restricting a constraint : unload/reload the concerned database fragment (e.g. a table) and discard the data that violate the new constraint.
- M consists in extending a domain : unload/reload the concerned data (e.g. table contents).
- M consists in adding a construct that the DBMS can accommodate dynamically (e.g. a new table, a new column, a new index) : no data change needed.
- M consists in adding a new construct that the DBMS can not accommodate dynamically : unload/reload the concerned database fragment and leave new structures empty.
- M is to remove a construct that the DBMS can discard dynamically (e.g. table or index) : no data change needed.
- The initial change is to remove a construct that the DBMS can not discard dynamically (e.g. a column) : unload/reload the concerned database fragment and skip the obsolete data.
- The initial change can be expressed as a reversible transformation : apply instance mapping **t** of the transformation to the concerned database fragment.

Three phenomena make the process a bit more complex :

1. When the change results in more than one schema modification, several of the cases mentioned above may occur simultaneously. The actions they trigger must be merged in order to define a unique conversion process. For instance, removing two attributes from an entity type can not induce two unload/reload processes.
2. If SO is not a final product, i.e. when other transformation-based processes were carried out on schema so (i.e. when sf $\neq$so), the concerned part of the schema may be involved in a chain of transformations Σ1, Σ2, ... Σn described in the history. Therefore, the composed instance mapping **tn**(..**t2**(**t1**(E))) must be applied on the concerned data E. Translating this composition into a unique conversion process can be complex.
3. The users can be interested in keeping obsolete data in the database. In this case, data versioning can be used, as in temporal database. For instance, [31] proposes an extension of schema evolution to schema versioning in order to support history of both the data and the schemas.

The reader interested in the various techniques of propagating structural changes to the instances can be referred to [26] for instance.

Step 3 - Changing the application programs

A change in the data structures may induce strong modifications in the application programs. Besides light changes like attribute domain extension, which most often implies modifying the format or the name of program variables, deeper modifications may occur, such as changing the cardinality of relationship type roles. If, from now on, an order can be passed by several customers, instead of by one only, there is a high probability of some procedural sequences being replaced by loop structures in the concerned programs.

When the programs have been built through transformational techniques [10] the same reasoning as above could be applied for their modification. However, this approach is even more unfrequent in this domain than in databases. Therefore we must consider the most

general case in which no formal description of the program is available. If we consider that reverse engineering complete application programs is still an unsolved issue, then we must admit that no complete solution can be developed in the general case.

The approach proposed is much more modest, though still far from trivial. It consists in detecting all the program sections that may have to be changed. These sections are those which are directly or indirectly involved in manipulating instances of the modified data structures. This analysis can be based on techniques such as *program slicing* that allow extracting the (forward and) backward influence domain of a program entity [5, 42]. Based on this information, the changes will be carried out manually, but it is possible to give the programmer hints on what kind of modifications are most probable.

6.2 Strategy 2 : Backward database maintenance

Here too, we make the hypothesis that the reference methodology, as well as the history of the database design are available, possibly through reverse engineering. The basic problem is to detect the reason for the technical change that has been applied on the DBMS schema (i.e. what requirements R changed), and to update the history accordingly in such a way that it now describes how the actual DBMS schema could have been produced from the conceptual specifications. To simplify the presentation, we will consider how modification of a relational schema can result into a new history.

The process starts with the elicitation of the exact changes that occurred. They can be obtained either by a differential analysis of both DBMS schemas, or by direct specification from their author. These changes can always be expressed as schema transformations. We first present how to process a change that results from a reversible transformation.

- *Table T has been partitioned vertically.*

  First, the reason why this update occurred, i.e. requirements R, has to be elicited in order to select the process instance p in which the update is the most relevant : ease of management, modularity, DBMS constraints (e.g. row length), performance, normalization, readability, etc.

  Then, we retrieve the object O in the output schema so of p that is the origin of T.

  Object O is then transformed into fragments O1, O2, .., that can be translated into table fragments T1, T2, ... This transformation is appended to the history of p. In all the following process instances of history, the processing of O is replaced by the processing of O1, O2, ... If O has not been deeply restructured in the origin history, this replacement can be automated. In some more complex cases however, the designer may have to check the validity and the completeness of the succeeding transformations. Indeed, in complex histories, these fragments may require additional processing that cannot be inferred from the history of the initial table.

This procedure is illustrated in figure 6.2.

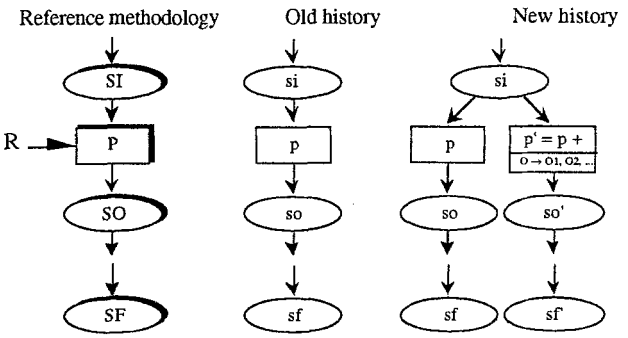

Fig. 6.2 - Building a new design following a semantics-preserving change of the logical schema. The relevant process P is identified, and its instance p is retrieved. A new version p' of p is built by adding a transformation that results in the observed change in sf'. The subsequent processes are replayed.

Some specific transformations, such as *semantics-augmenting* ones, can be processed by simpler, ad hoc alternate procedures. Here are some examples.

- *A new table has been added to the schema.* The abstract representation of the table (logical entity type) and of each of its columns (logical attributes) are added to the conceptual schema[17]. Through reverse engineering techniques, these objects can be conceptualized into a new entity type, a new relationship type, or a new multivalued attribute for instance. This conceptualization process uses reversible transformations that generate a reversible backward mapping. The initial history can be modified accordingly by distributing the reverse transformations in the existing design process instances (see strategy 3).
- *A new column has been added to a table.* The abstract representation of the column (a logical attribute) is added to the conceptual object that is mapped to the table (determined through the backward mapping). This attribute is then reverse engineered to recover its conceptual interpretation. This process can be deterministic provided it is controlled by the author of the update. For instance, this logical attribute can be conceptualized into a relationship type or a mere conceptual attribute. The initial history is modified accordingly.

Converting the database contents and the application programs can be done easily as proposed in steps 2 and 3 of strategy 1.

6.3 Strategy 3 : Database reverse engineering

The DB-MAIN approach is based on the two-phase process that has been developed in the PHENIX [14, 17, 21] and REDO projects [34].

- The first phase, the *data structure extraction process*, consists in recovering the DBMS schema from the source code of the applications (and from other sources, such the database contents) and in expressing them in the generic model. This process can be very complex since many structural and constraint specifications are not explicitly expressed in data structures, but are burried, duplicated and spread throughout the application programs.
- The second phase, called *data structure conceptualization*, is intended to clean the data structures by eliminating the non-semantic constructs and by recovering the initial concepts translated into constructs according to the DBMS model. This process is based on transformational techniques such as those that have been described in [18]. Thanks to them, we are provided with a formal backward mapping of the DBMS schema to the conceptual schema. Since these transformations are reversible, we can easily derive the forward mapping that explains how the recovered conceptual schema can be translated into the actual DBMS schema. The work is not quite completed yet, since we need to recover a history for the database, not only in terms of primitives, but also of design processes instances.

  Thefore, we have to choose a reference methodology, e.g. close to the way the database may have been built in the past. From this methodology, we derive a possible history, made of empty design process instances. The primitive instances that derive from the conceptualization phase are then distributed into these different design process instances (knowing which requirements the application were to satisfy is a strong hint here). See fig. 6.3.

Finally, we are provided with a **hypothetical database design history** which enjoys a outstanding property : when replayed on the recovered conceptual schema, it produces the actual database structure according to the reference database design methodology.

From now on, we can apply strategies 1 and 2 to make this database evolve.

---

[17] Mixing logical and conceptual constructs in the same schema can be questioned. In fact, it is a common practice in reverse engineering, where some parts of the schema have already been conceptualized, while other parts still are uninterpreted, i.e. at the logical level. Hence the importance of a common generic model.

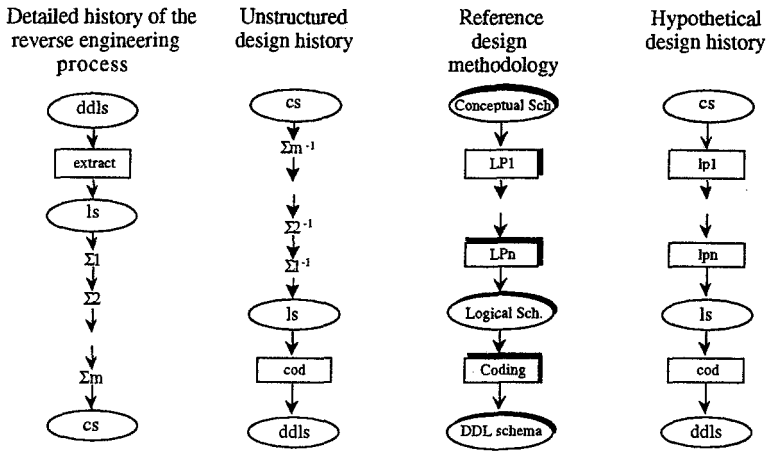

Fig. 6.3 - Design recovery through reverse engineering. The logical schema `ls` *is retrieved from the DDL source text* `ddls` *by* `extract`*, an instance of the Data structure extraction process, and transformed by successive reversible transformations Σ1, Σ2, .., Σm into the conceptual schema* `cs`*. Reversing these transformations (the inverse of Σi is noted* Σi^{-1}*), we obtain an unstructured, hypothetical history. The latter is mapped into the chosen reference design methodology to form the hypothetical design history.*

7. The DB-MAIN CASE Tool

The DB-MAIN database engineering environment (DB-MAIN CASE tool, for short) is to be used as an experimental research laboratory, as an industrial tool, and as an educational tool as well. Just like any CASE tool, DB-MAIN offers the standard functions for specifications entry, management, consultation, validation, storing and reporting. It also offers generators for some standard DBMS. The remaining of this section will instead put forward the original characteristics that are intended to satisfy the three objectives evoked hereabove and to support, in addition to the mandatory forward engineering activities, the maintenance activities.

- The **repository** contains the representation of the history of a project, together with the version of the methodology used to conduct the project. It includes the precise description of all the schemas, the processing specifications, the design processes that appeared during the course of the project. The repository is an OO-ER database (*à la* OMT), implemented by a C++ object base, in which each specification item is represented by an instance of an object (SCHEMA, ENTITY-T, REL-TYPE, ROLE, ATTRIBUTE, etc). Each instance can have various specialized annotations that describe, a.o. its semantics. The histories are implemented as follows. The reference methodology is translated (through a compiler) into a set of instances of objects PRODUCT, PROCESS and ACTIONS (detailing strategies). Product instances are mainly made of data structures (represented by, e.g., instances of SCHEMA). Process instances are represented by instances of PROCESS_INSTANCE when they derive from activation of *design processes* and by journal transactions when they are activations of *primitives*. Hypotheses, decisions and justifications are specialized *annotations* that can be pinned on any object.
- DB-MAIN includes sophisticated **transformational toolkits** that allow manipulating the specifications with semantics-preserving operators. The reversibility of most of these operators is essential when performing reverse engineering and backward maintenance.
- A **methodological engine**, driven by the reference methodology stored in the repository, allows for the customization of the CASE tool according to a specific methodology. This engine has two roles. The first one is the customization of the tool to the exact needs of the development staff and to the corporate culture. Secondly, it provides, through history recording, a formal tracing of the design activities that allows an *a posteriori* examination

and analysis of these activities. In particular, it includes an intelligent **replay processor** that is able to replay, with some variations, selected activities of a former history.

- An **assistant** offers high-level functions that carry out complex or tedious operations (e.g. global schema transformation, name processing, code generation) and support heuristics (normalization, optimisation, program analysis).
- The *data structure extraction* process of **reverse engineering** is supported by source code parsers that load in the repository the abstract expression of explicit data structures decoded in application programs and DDL texts. A complementary pattern-matching engine allows sophisticated text analysis to elicit implicit structures, particularly in procedural code (e.g. SQL triggers, COBOL sections, etc).
- The tool includes a repository-based 4GL that provides for **functional extensibility** without resorting to C++ programming. It allows seamlessly adding functions such as specific transformations, complex heuristics, evaluators, non standard loaders and generators.

A first prototype version of the DB-MAIN CASE tool is already available. This version runs on PC compatible, MS-Windows 3.1, workstations. It includes basic functions for database forward and reverse engineering, a basic transformation toolkit, parsers for IDS-2, COBOL and SQL source texts, an assistant in schema processing, journaling and a replay processor for primitives[18]. Though it is far from including all the components described hereabove, it is intensively used for educational purpose as well as in several large database applications.

8. Conclusions

We have proposed four essential strategies (and describe three of them) that should bring a considerable help in making complex, data-centered, applications evolve according to the changes that naturally occur in the requirements.

Supporting these strategies requires models that can describe both the specifications at all abstraction levels, and the activities that process these specifications throughout the design of the database and its applications. These models, presented in sections 3 to 5, all have a formal basis (though this aspect has not been developed in this paper; see [12, 13] for instance) that allows deriving important properties of a collection of specifications. For example, the reversibility property of the transformations makes possible the construction of a hypothetical design history from the history of the reverse engineering process.

It is worth to stress once more the genericity of these models. Indeed, these models address all the levels of abstraction in the specifications, and the whole spectrum of determinism in design behaviours. This genericity provides a high level of independence of the proposals.

The CASE tool inherits this genericity, which allows building functionalities (e.g. transformations) that can be used at any abstraction level.

Some important aspects have not been included in the scope of the project . For instance, DB-MAIN does not address the formal description of the requirements and their processing in the process model. Requirements must be taken into account in the elaboration of the process model, and in particular in the definition of the products and of the strategies of the design processes. So far, nothing else is planned to help the designer to control the satisfaction of the requirements (see [6] and [23] for instance). As a consequence, documenting hypotheses and decisions is only supported through informal and passive annotations. The same can be said of time-related and resource aspects, that would allow project management.

However, the process model enjoys three important properties : (1) it allows a precise and natural specification of both standard and empirical design behaviour, including reverse

---

[18] An educational version of the tool along with various case studies can be obtained through FTP : connect to `info.fundp.ac.be`; login : `anonymous`; password : "your e-mail address"; `cd pub/projects/db_main`; `get README`; `get dbm_386.exe`; execute the latter on a PC to obtain all the basic components of DB-MAIN. In addition, a (draft) tutorial is available as `tutorial.exe`, to execute on a PC. Please contact `jlh@info.fundp.ac.be` in case of problems.

engineering and maintenance activities themselves, (2) it forms a sufficient framework to monitor efficiently the four maintenance strategies presented in section 2, (3) it is fairly easy to implement into standard window-based environment.

9. REFERENCES

[1] Andany, J., Léonard, M., Palissier, C., *Management of Schema Evolution in Databases*, in Proc. of the 17th Int. Conf. on VLDB, Morgan-Kaufmann, pp. 161-170, 1991

[2] Balzer, R., *Transformational implementation : An example*, IEEE TSE, Vol. SE-7:1, 1981

[3] Batini, C., Ceri, S., Navathe, S., *Conceptual Database Design*, Benjamin/ Cummings, 1992

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

[5] Biggerstaff, T., J., Mitbander, B., G., Webster, D., *The Concept Assignment Problem in Program Understanding*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, May 1993, IEEE Computer Society Press, 1993

[6] Chung, L., Katalagarianos, P., Marakakis, M., Mertikos, M., Mylopoulos, J., Vassiliou, Y., *From information system requirements to designs : A mapping framework*, Information Systems, Vol. 16, pp. 429-461, 1991

[7] Davis, K., H., Arora, A., K., *Converting a Relational Database model to an Entity Relationship Model*, in Proc. of Entity-Relationship Approach : a Bridge to the User, 1988

[8] Desclaux, C., Ribault, M., Cochinal, S., *RE-ORDER : A Reverse engineering Methodology*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December, pp. 517-529, EC2 Publish. 1992

[9] Ewald, C., A., Orlowska, M., E., *A Procedural Approach to Schema Evolution*, in Proc. of CAiSE'93, Springer-Verlag, 1993

[10] Fikas, S., F., *Automating the transformational development of software*, IEEE TSE, Vol. SE-11, pp1268-1277, 1985

[11] Fonkam, M., M., Gray, W., A., *An approach to Eliciting the Semantics of Relational Databases*, in Proc. of 4th Int. Conf. on Advance Information Systems Engineering - CAiSE'92, pp. 463-480, May, LNCS, Springer-Verlag, 1992

[12] Hainaut, J.-L., *A Generic Entity-Relationship Model*, in Proc. of the IFIP WG 8.1 Conf. on *Information System Concepts: an in-depth analysis*, North-Holland, 1989

[13] Hainaut, J-L., *Entity-generating Schema Transformation for Entity-Relationship Models*, in Proc. of the 10th Entity-Relationship Approach, San Mateo (CA), 1991

[14] Hainaut, J-L, *Database Reverse Engineering, Models, Techniques and Strategies*, in Preproc. of the 10th Conf. on Entity-Relationship Approach, San Mateo (CA), 1991

[15] Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O., *Database CASE Tool Architecture : Principles for Flexible Design Strategies*, in Proc. of the 4th Int. Conf. on Advanced Information System Engineering (CAiSE-92), Springer-Verlag, LNCS, 1992

[16] Hainaut, J-L., Cadelli, M., Decuyper, B., Marchand, O., *TRAMIS : a transformation-base database CASE tool*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December 1992, EC2 Publish., 1992

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

[18] Hainaut, J-L, Chandelon M., Tonneau C., Joris M., *Transformational techniques for database reverse engineering*, in Proc. of the 12th Int. Conf. on ER Approach, Arlington-Dallas, ER Institute/ Springer-Verlag, 1993

[19] Jarke, M., et al., *DAIDA : An environment for evolving information systems*, ACM Trans. on Information Systems, Vol. 10, Jan. 1992

[20] Jarke, M., et al., *NATURE : Requirements Engineering : An Integrated View of Representation, Process and Domain*, NATURE Report Series, No. 93-07, available from <natrep@informatik.rwth-aachen.de>

[21] Joris, M., Van Hoe, R., Hainaut, J-L., Chandelon M., Tonneau C., Bodart F. et al., *PHENIX : methods and tools for database reverse engineering*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December 1992, EC2 Publish., 1992

[22] Kozaczynsky, Lilien, *An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation*, in Proc. of Entity-Relationship Approach, 1987

[23] Mylopoulos, J., Chung, L., Nixon, B., *Representing and Using Nonfunctional requirements : A Process-Oriented Approach*, IEEE TSE, Vol. 18, No. 6, June 1992

[24] Navathe, S., B., *Schema Analysis for Database Restructuring*, in ACM TODS, 5:2, 1980

[25] Navathe, S., B., Awong, A., *Abstracting Relational and Hierarchical Data with a Semantic Data Model*, in Proc. of Entity-Relationship Approach : a Bridge to the User, 1988

[26] Nguyen, G., T., Rieu, D., *Schema evolution in object-oriented database systems, Data & Knowledge Engineering*, 4 (1989) pp. 43-67, North-Holland

[27] Nilsson,E., G., *The Translation of COBOL Data Structure to an Entity-Rel-type Conceptual Schema*, in Proc. of Entity-Relationship Approach, October, 1985

[28] Potts, C., Bruns, *Recording the Reasons for Design Decisions*, in Proc. ICSE, IEEE, 1988

[29] W.J. Premerlani, W., J., Blaha, M.R., *An Approach for Reverse Engineering of Relational Databases*, in Proc. of the IEEE Working Conf. on Reverse Engineering, Baltimore, May 1993

[30] Roddick, J., F., *Schema Evolution in Database Systems - An Annotated Bibliography*, SIGMOD Record, Vol. 21, No. 4, pp. 35-40, Dec. 1992

[31] Roddick, J., F., Craske, N., G., Richards, T., J., *A taxonomy for Schema Versioning Based on the Relational and Entity-Relationship Models*, in Proc. of the 12th Int. Conf. on ER Approach, Arlington-Dallas, ER Institute, 1993

[32] Rolland, C., *Modeling the Requirements Engineering Process*, in Proc of the 3rd European-Japanese Seminar in Information Modeling and Knowledge Bases, May 1993, Budapest (preprints)

[33] Rosenthal, A., Reiner, D., *Theoretically sound transformations for practical database design*, in Proc. of Entity-Relationship Approach, 1988

[34] Sabanis, N., Stevenson, N., *Tools and Techniques for Data Remodelling Cobol Applications*, in Proc. 5th Int. Conf. on Software Engineering and Applications, Toulouse, 7-11 December, pp. 517-529, EC2 Publish. 1992

[35] Schneiderman, B., Thomas, G., *An architecture for Automatic Relational Database System Conversion*, ACM TODS, Vol. 7, No. 2, pp. 235-257, 1982

[36] Souquières, J., Lévy, N., *Description of Specification Developments*, in Proc. of Int. Conf. on Requirements Engineering, San Diego, IEEE, 1993

[37] Springsteel, F., N., Kou, C., *Reverse Data Engineering of E-R designed Relational schemas*, in Proc. of Databases, Parallel Architectures and their Applications, March, 1990

[38] Teorey, T. J., *Database Modeling and Design*, Morgan Kaufmann, 1990

[39] Tsuda, K., Yamamoto, K., Hirakawa, M., Tanaka, M., Ichikawa, T., *MORE : An Object-Oriented Data Model with a Facility for Changing Object Structure*, IEEE Trans. on Knowl. and Data Eng., Vol. 3, No. 4, Dec. 1991

[40] van Bommel, P., *Database Design Modifications based on Conceptual Modelling*, in Proc. of the 3rd European-Japanese Seminar on Information Modelling and Knowledge Bases, May 1993, Budapest, pp. 276-288 (Preprint)

[41] Van Zuylen, H., *The REDO Handbook - A compendium of reverse engineering for Software Maintenance*, REDO Project report 2487-TN-WL-1027, Nov. 1991

[42] Weiser, M., *Program Slicing*, IEEE TSE, Vol. 10, 1984, pp 352-357

[43] Winans, J., Davis, K., H., *Software Reverse Engineering from a Currently Existing IMS Database to an Entity-Relationship Model*, in Proc. of Entity-Relationship Approach : the Core of Conceptual Modelling, pp. 345-360, October, 1990