Database Design Recovery[1]

J-L. Hainaut, J. Henrard, J-M. Hick, D. Roland, V. Englebert[2],

ABSTRACT

The design of a software component, such as a database, is the trace of all the processes, products and reasonings that have led to the production of this artifact. Such a document is the very basis of system maintenance and evolution processes. Unfortunately, it does not exist in most situations. The paper describes how the design of a database or of a collection of files can be recovered through reverse engineering techniques. Recording the reverse engineering activities provides a history of this process. By normalizing and reversing this history, then by conforming it according to a reference design methodology, one can obtain a tentative design of the source database. The paper describes the baselines of the approach, such as a wide spectrum specification model, semantics-preserving transformational techniques, and a design process model. It describes a general procedure to build a possible DB design, then states the requirements for CASE support, and describes DB-MAIN, a prototype CASE tool which includes a history processor. *Finally it illustrates the proposals through an example.*

KEYWORDS

design recovery, database reverse engineering, database evolution, transformational approach, process modelling, CASE tool

1. Introduction

> ***[...] design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, [...].***
>
> [BIGG,89]

History and Design

The history of a complex process, whatever its nature and its objective, is the trace of all the operations that have been carried out in order to complete it. This definition stands for program execution for exemple, but it is also applicable to human activities, such as information system engineering. The *design* of an information system, or of a part thereof, is the history of the processes through which the system was built, together with the various specification products (schemas, documentation, operational code) used and produced by these processes. The design also includes the requirements that the system was to satisfy and the reasonings (also called the *rationale*) according to which the processes and the decisions were carried out. The design of an existing system should be available, not only as the core documentation, but also as the basis for further system

[1] This research is a part of the DB-MAIN project, which is partly supported by ACEC-OSI (Be), ARIANE-II (Be), Banque UCL (Lux), BBL (Be), Centre de recherche public H. Tudor (Lux), CGER (Be), Cockerill-Sambre (Be), CONCIS (Fr), D'Ieteren (Be), DIGITAL (Be), EDF (Fr), EPFL (CH), Groupe S (Be), IBM (Be), OBLOG Software (Port), ORIGIN (Be), Ville de Namur (Be), Winterthur (Be), 3 Suisses (Be). The DB-PROCESS subproject is supported by the *Communauté Française de Belgique.*

[2] Institut d'Informatique, University of Namur, rue Grandgagnage, 21 - B-5000 Namur - jlh@info.fundp.ac.be

maintenance and evolution. Ideally, this design should be recorded in the repository of a CASE tool which supports these maintenance and evolution processes.

The concept of *history* is broader than suggested above. Indeed, any system engineering activity can be described by its history. Such is the case for system reverse engineering, system migration, system evolution, and system integration to mention only some examples. In the following, we will often make use of the more general term *history* instead of *design.*

The role of histories in system evolution

One immediate application of a history is to *explain* how and why the system has been developed in this way some years ago. Another important usage of a history is its *replay*. For instance, let us consider that we are provided with the history of the development of a database. This history is supposed to be the trace of all the activities of the conceptual design, of the logical design, of the physical design, and to include all the schemas which have been elaborated during these activities. Let us now suppose that we have to make a minor change in the conceptual schema of a database, such as adding a simple attribute. Ideally, all we should have to do is to carry out this change in the conceptual schema, then to *replay* the former history. If this history has been recorded in the repository of a CASE tool, this second operation can be carried out automatically, and can produce new versions of the logical and physical schemas. Of course, if the change is deeper, merely replaying the former design activities could prove insufficient since the conceptual change could imply a revision of the former decisions. The concept of history replay must be refined in order to cope with these more complex situations. Anyway, considering both applications of histories, it appears clearly that this concept must play a central role in computer-supported system evolution [BIGG,89] [HAI,94].

Unfortunately, in the current state of development practices, all this discussion may appear as basically theoretic, and far from the reality. Indeed, most systems have been (and still are) designed without any methodological concern, and consequently without CASE support. Consequently, many systems have no decent documentation, and when it exists and is up-to-date and usable, the history is unknown. In conclusion the design of most legacy systems is absent, or, at best, degenerates into a collection of hopefully correct and consistent schemas.

Reverse engineering

Trying to make a legacy information system evolve implies first to recover a possible design, i.e. a possible history together with its various (inherited or recovered) schemas and other documents. When no documentation is available for such a system, it is possible to rebuild it, at least partially, through a process called *reverse engineering*. For instance, successfully reverse engineering an existing database or a collection of files (i.e. the *physical* system) yields plausible logical and conceptual schemas for this database. It can be shown [HAI,93b] [HAI,95c] that most system forward and reverse engineering activities are transformational by nature, i.e. that they can be modelled as chains of schema transformations. Indeed, they most generally appear as processes which transform one specification product into another one. In addition reverse engineering can be described as a suite of processes, each of which is the inverse of a definite forward engineering process [HAI,93a].

Design recovery

We can record the activities which are carried out during reverse engineering. This history will be made of a sequence of elementary transformations. This is a free, but extremely valuable by-product of the process. Indeed, reversing this history, i.e. reversing the order of the operations, and replacing each of them by its inverse, should provide a way to get back the physical system from the abstract specification just recovered through the reverse engineering process. In this perspective, reverse engineering seems to be a realistic way to recover a possible design of the system.

This is precisely the idea developed in this paper, applied to database design recovery.

About this paper

The paper is organized as follows. Section 2 shortly describes the data structure specification model which will be used to illustrate the concepts developed. Section 3 defines the concept of schema transformation. Section 4 discusses the notion of history and relates it to that of methodology. Section 5 proposes a general methodology for database reverse engineering. Section 6 presents the principles of database design recovery. In section 7 we discuss the role of CASE technology in design recovery, while an example is developed in section 8.

2. A data structure specification model

Database design and reverse engineering are concerned with building, converting and transforming database schemas at different levels of abstraction, and according to various paradigms. Elaborating DMS-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 presentation, 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 wide spectrum schema specification model. This model and its transformational operators are intended,

- to support forward as well as reverse engineering,
- to express conceptual, logical and physical schemas, as well as their manipulation,
- to support any DMS model and the production and manipulation of their schemas.

In short, conceptual schemas as well as physical schemas are expressed into a unique, generic, *extended entity-relationship model.*

In this model, a *schema* is a description of the data structures to be processed. It is made up of specification constructs which can be classified into the usual three abstraction levels, namely conceptual, logical and physical (*Fig. 1*). We will enumerate the main concepts in each level :

conceptual constructs

entity types (with/without attributes; with/without identifiers), super/subtype hierarchies (single/multiple inheritance, total and disjoint properties), relationship types (binary/N-ary; cyclic/acyclic), roles of relationship type (with min-max cardinalities; with/without explicit name; single/multi-entity-type), attributes (of entity and relationship types; multi/single-valued; atomic/compound), identifiers (of entity type, relationship type, attribute; comprising attributes and/or roles), constraints (inclusion, exclusion, coexistence, at-least-one, etc);

logical constructs

record types, fields, foreign keys, redundancy, etc;

physical constructs

files, access keys (abstracting index, calc key, etc), physical data types, bag and list multivalued attributes, and other implementation details.

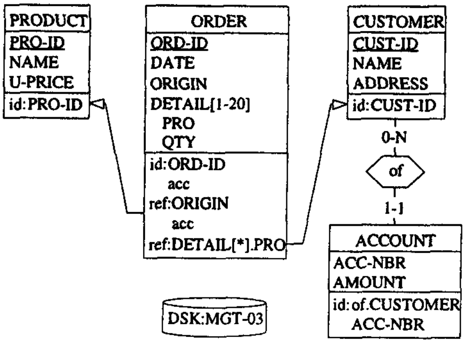


Figure 1 - A typical hybrid schema that could appear during reverse engineering. This schema includes conceptual objects (PRODUCT, CUSTOMER, ACCOUNT, of), logical objects (record type ORDER, with single-valued and multivalued foreign keys) and physical objects (access keys ORDER.ORD-ID and ORDER.ORIGIN; file DSK:MGT-03).

In database engineering, a schema describes a fragment of the data structures at a given level of abstraction. In reverse engineering, an *in progress* schema may even include constructs at different levels of abstraction. *Fig. 1* illustrates this fact through a schema which includes conceptual, logical and physical constructs. Ultimately, this schema will be completely conceptualized through processing of the remaining logical and physical constructs.

Note
A schema can be defined as a set of constructs, or specification objects. Therefore, set-theoretic relations and operators apply on schemas. For instance, a schema can be declared a subset of another one ($S_2 \subseteq S_1$) or can be defined as the union of other schemas ($S_3 = S_1 \cup S_2$). However, a proper subset of a valid schema is not necessarily a valid schema : a relationship type with one role only can be a subset of a valid relationship type, but is not valid itself.

3. Schema transformation

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. Several authors postulate 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 [NAVA,80] [KOBA,86] [KOZA,87] [ROS,88] [BATINI,92] [HALPIN,95] [RAUH,95]. Moreover, some authors claim that the whole database design process, together with other related activities, can be described as a chain of schema transformations [BATINI,93] [HAI,93b] [HAI,95c][3].

[3] It is interesting to note that this approach has long be considered in pure software development. According to [BALZER,81] and [FICKAS,85] for instance, *the process of developing a program [can be] formalized as a set of transformations.*

Schema transformations are essential to formally define forward and backward mappings between schemas, and particularly between conceptual structures and DBMS constructs. In addition, they provide a concise and precise trace of any engineering activity, and will be the building blocks of histories.

The notion has been defined in [HAI,91] and [HAI,95c], and can be summarized as follows.

3.1 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** specifies 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** specifies 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>**.

3.2 Reversibility

In terms of semantics-preservation, some transformations appear to augment the semantics of the source schema (e.g. adding an attribute), some remove semantics (e.g. removing 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*. If a transformation is reversible, then the source and the target schemas have the same descriptive power, and describe the same universe of discourse, although with a different presentation (or syntax).

- A transformation Σ1 = <P1,Q1,t1> = <T1,t1> is *reversible*, *iff* there exists a transformation Σ2 = <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]). Σ2 is the inverse of Σ1, but, surprisingly, not conversely. For instance, an arbitrary instance c' of T(C) may not satisfy the property c'=t1(t2(c')).
- If Σ2 is reversible as well, then Σ1 and Σ2 are called *symmetrically reversible*. In this case, Σ2 = <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. 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, [BALZER,81] proposes the concept of *correctness-preserving* transformation aimed to compilable and efficient program production.

Remark.

We have discussed the concept of reversiblity in a context in which some kind of instance equivalence is preserved. However, the notion of inverse transformation is more general. Any transformation, be it semantics-preserving or not, can be given an inverse. For instance, del-ET(CUSTOMER), which removes entity type CUSTOMER from its schema, clearly is not a semantics-preserving operation, since its mapping t has no inverse. However, it has an inverse transformation, namely create-ET(CUSTOMER). Since only the T part is defined, this partial inverse is called a *structural inverse* transformation. We will discuss these operators in more detail in the next sections.

3.3 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 [HAI,92]. 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, figure 2 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)). The t part of the transformations will be ignored from now on.

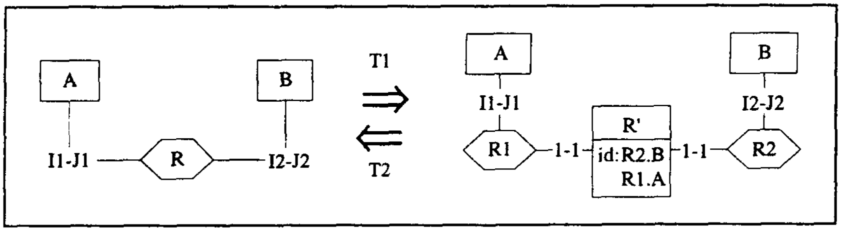


Figure 2 - Representation of structural mapping T1 (from left to right) & T2 (from right to left) of a typical SR-transformation.

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

3.4 Stuctural analysis of a transformation

A transformation is known to replace construct C with construct C' in schema S, to yield new schema S'. The effect of transformation T in schema S can be precised as follows. Let us consider the structural functions C_-, C_+ and C_0 :

$C_-(T) = S - S'$ returns the objects of S that have disappeared;

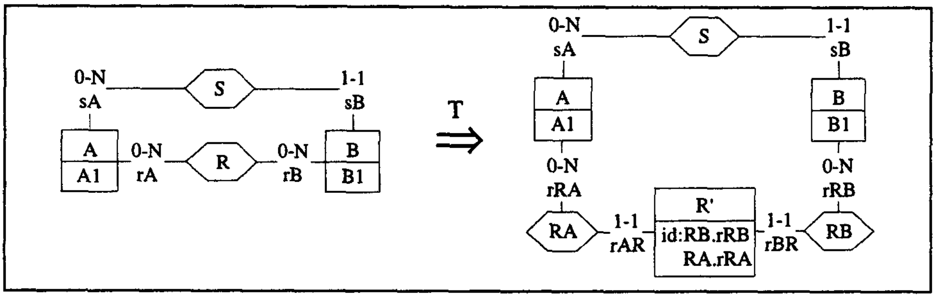
$C_+(T) = S' - S$ returns the new objects that appear in S';

$C_0(T)$ returns the objects of S that are concerned by T, but that are preserved from S to S' (the *catalytic* constructs of T).

We also have :

$$C(T) = C_0(T) \cup C_-(T)$$
$$C'(T) = C_0(T) \cup C_+(T)$$
$$S' = (S - C_-(T)) \cup C_+(T)$$

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



The structural functions evaluate as follows :

$C_-(T)$ = {R,rA,rB}
$C_+(T)$ = {R',RA,RB,rRA,rAR,rRB,rBR,id(R')}
$C_0(T)$ = {A,B}
S = {A,B,A1,B1,S,sA,sB,R,rA,rB}
S' = {A,B,A1,B1,S,sA,sB,R',RA,RB,rRA,rAR,rRB,rBR,id(R')}
C(T) = {A,B,R,rA,rB}
C'(T) = {A,B,R',RA,RB,rRA,rAR,rRB,rBR,id(R')}

3.5 Signature of a transformation

In a history, a transformation will be specified through its **signature**, that states the name of the transformation, the names of the concerned objects in the source schema, and the names of the new objects in the target schema. For example, the signatures of the transformations T1 and T2 in Fig. 2 are as follows[4] :

T1 : (R',{(A,R1),(B,R2)}) ← RT-to-ET(R)

T2 : R ← ET-to-RT(R')

The first one is interpreted as "*when applying RT-to-ET to relationship type R, the new entity type is called R', the rel-type involving A is called R1 and that involving B is called R2*". The second one must read as follows : "*when applying ET-to-RT to entity type R', the new rel-type is called R*". The objects which are involved in the operation, but that can be identified in the schema from the names mentioned in the signature, are not specified. In the signature of T2 for instance, entity types A and B are not mentioned since they can be deduced as "*all the entity types linked to R' in the source schema*". A signature alone does not comprise the C_-, C_+ and C_0 structural components, but it can be used to identify them in the source and target schemas. In addition, the format of a signature is not unique, but depends, a.o., on the default naming conventions. For instance, the roles are given default names in transformations T1 and T2 described above.

[4] Fixed-length lists are enclosed into parentheses, while variable-length lists are enclosed into curly brackets.

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

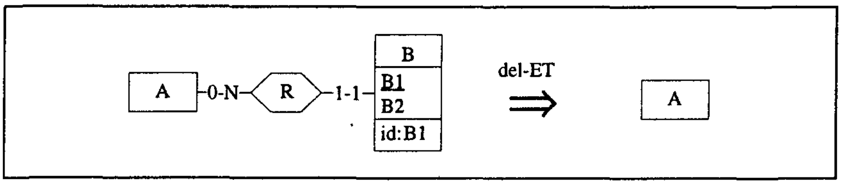
```
(R',{(A,R1),(B,R2)}) ← RT-to-ET(R)
```

could be instantiated, in an actual schema, into

```
(ORDER,{(CUSTOMER,from),(PRODUCT,of)}) ← RT-to-ET(order)
```

From these examples, we can observe an essential property of the signatures : their *reversibility*. Being provided with the right-side schema and the signature of T2, we can derive the signature of T1, and conversely. In other words, the signature provides enough information, not only for *redoing* the operation, but also to *undo* it.

This property is less obvious for some non-SR-transformations. Let us consider the example of the `del-ET` operator, which removes an entity type from a schema. It can be illustrated as follows.



At first glance, it seems that the following signature could be quite right :

```
() ← del-ET(B)
```

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

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

```
() ← del-ID(B,{B1},δ)
() ← del-Att(B,B1,δ)
() ← del-Att(B,B2,δ)
() ← del-Role(R,B,δ)
() ← del-Role(R,A,δ)
() ← del-RT(R,δ)
() ← del-ET(B,δ)
```

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

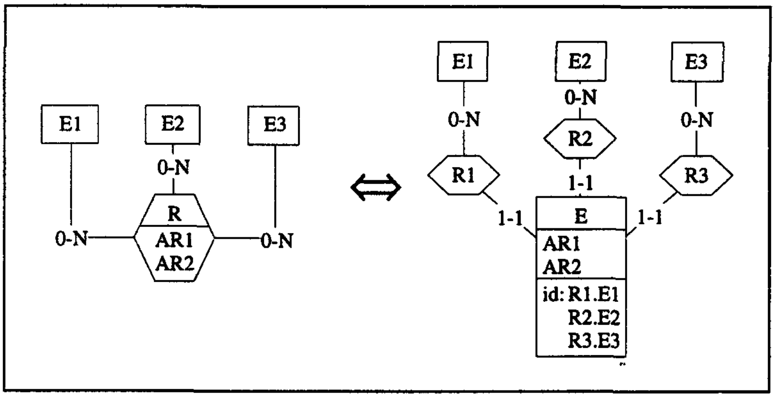
3.6 Some examples of reversible transformations

A realistic toolset aimed at building complex database schemas would include several dozens of schema transformations. We will briefly specify some of the most popular techniques for which only the graphical expression of mapping T will be given. Some examples of usage are proposed, together with the signature of the direct and inverse transformations. A more comprehensive development will be found in [HAI,91], [HAI,93b] and [HAI,95c] for instance.

3.6.1 Transformation of a relationship type into an entity type

Any relationship type R can be expressed by an entity type E, and as many one-to-many relationship types R1, R2, .., as there are roles in R. The transformation preserves the cardinality constraints, the possible attributes as well as the identifiers. It is an SR-transformation.

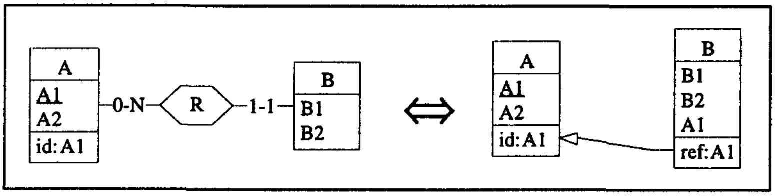
This generalization of the transformation depicted in *Fig. 2* will be used to reduce complex relationship types, such as those with more than 2 roles, or with attributes. It can also be used to get rid of many-to-many or recursive relationship types.



Signatures : `(E,{(E1,R1),(E2,R2),(E3,R3)})` ← `RT-to-ET(R)` *(direct)*
`R` ← `ET-to-RT(E)` *(inverse)*

3.6.2 Transformation of a relationship type into a foreign key

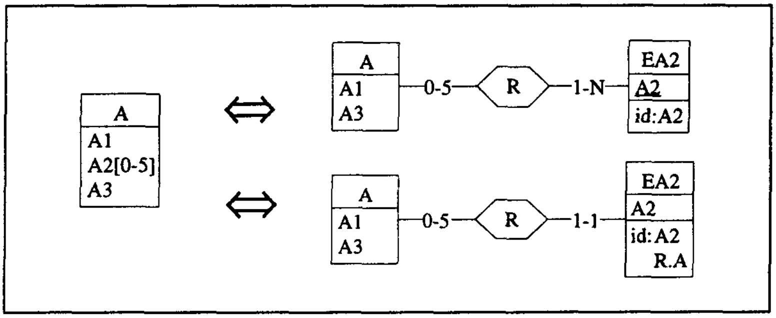
This one replaces a one-to-many relationship type R with a *foreign key*, which can be either multivalued or single-valued according to the cardinality of R. It is the main operator to produce relational schemas, but also standard file structures.



Signatures : `{A1}` ← `RT-to-FK(R,B)` *(direct)*
`R` ← `FK-to-RT(B,{A1},A)` *(inverse)*

3.6.3 Transformation of an attribute into an entity type

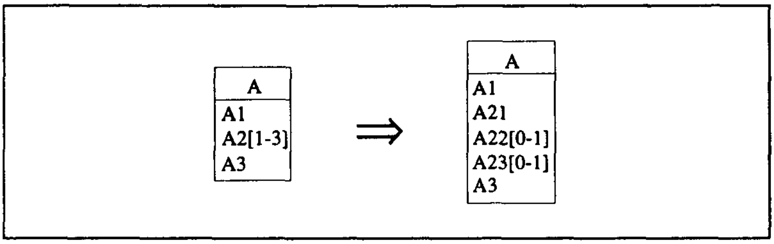
Transforming an attribute into an entity type is another common technique. It comes in two variants, namely **value representation** (above), in which each distinct value of A2, whatever the number of its instances, is represented by a distinct EA2 entity, and **instance representation** (below), in which each instance of A2 in each E entity is represented by a distinct EA2 entity. There are many applications of these techniques : eliminating multivalued attributes, eliminating optional attributes, eliminating compound attributes, extracting long attributes, defining dictionaries, promoting attributes (in conceptual analysis), normalizing entity types, etc.



Signatures : (EA2,R) ← Att-to-ET/Value(A,A2) *(direct)*
(EA2,R) ← Att-to-ET/Instance(A,A2) *(direct)*
A2 ← ET-to-Att(EA2) *(inverse)*

3.6.4 Transformation of a multivalued attribute into serial attributes

This technique is often used to represent multivalued attributes while avoiding building an additional table, which would in turn imply costly joins in application programs. Each instance of A2 is represented by a single-valued attribute. Unfortunately, this is an R-transformation only. Indeed, the right version will allow situations which are not permitted by the left schema. For instance, the uniqueness of the A2 values is no longer ensured (the serial attributes implement a bag and not a set) and an implicit order is defined on the value set.



Signatures : {A21,A22,A23} ← MultiAtt-to-SerialAtt(A,A2) *(direct)*
A2 ← SerialAtt-to-MultiAtt(A,{A21,A22,A23}) *(inverse)*

4. Methodologies and histories

4.1 Modelling methodologies and histories

A history is not an arbitrary sequence of operations. It should obey to a structured way of proceeding called a *methodology*[5]. A methodology specifies the products and the processes that appear when carrying out any instance of a general engineering activity. For instance, several database design methodologies have been proposed so far, ranging from standard ERA-based approches to the more recent OO approaches. Less common activities such as database reverse engineering [HAI,93a] or schema integration [SPACCA,92] have been formally described through specific methodologies as well.

Describing methodologies relates to (*software/design*) *process modelling*, a discipline that is concerned with the understanding, representation and computer-based support of the software engineering activities [POTTS,88], including the development of data structures in data-centered applications. Such activities can be modeled as a set of documents, or products (schemas, programs, specifications, etc) and a set of engineering processes that transform input products into output products according to specific requirements to satisfy. For instance, in the database realm, conceptual user's views are integrated into the 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 decomposed into a local set of products and processes, until primitive processes can be described.

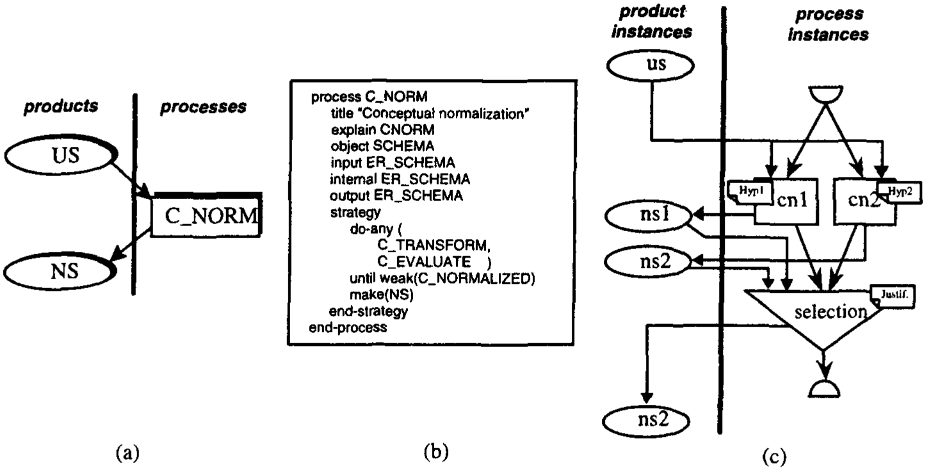
The model we have developed derives from proposals such as [POTTS,88] and [ROLL,93], extended to all database engineering activities. This model describes quite adequately not only standard design methodologies, such as the Conceptual-Logical-Physical approaches [TEOREY,94] [BATINI,92] but also any kind of heuristic design behaviours, including those that occur in reverse engineering. A fairly comprehensive specification of the model has been given in [HAI,94], but we will shortly recall the elements which will be used in the following (*Fig. 3*).

Product and product instance. A product instance is any outstanding specification object that can be identified in the course of a specific design. A conceptual schema, an SQL DDL text, a COBOL program, an entity type, a table, 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.

Process and process instance. A process instance is any logical unit of activity in a history which transforms 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 engineering processes and primitives. An *engineering process* is a goal-oriented process that is intended to make its input product satisfy specific 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

[5] We should have used the term *method*.

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.



```
process C_NORM
   title "Conceptual normalization"
   explain CNORM
   object SCHEMA
   input ER_SCHEMA
   internal ER_SCHEMA
   output ER_SCHEMA
   strategy
      do-any (
            C_TRANSFORM,
            C_EVALUATE  )
      until weak(C_NORMALIZED)
      make(NS)
   end-strategy
end-process
```

Figure 3 - Graphical representation of some aspects of the process model. (a) a process C_NORM is defined as a transformation of product US (un-normalized schema) into NS (normalized schema). (b) this text specifies the 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"). (c) a history that shows two instances cn1 and cn2 of C_NORM, carried out on product instance us, and yielding instances 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. The histories of cn1 and cn2 would show instances of C_TRANSFORM and C_EVALUATE.

Process strategy. The strategy of a process is the specification of how its goal can be achieved, i.e. how the process must be carried out. A strategy can be deterministic, in which case it reduces to an algorithm (and can often be implemented as a primitive), or it can be non-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* (should be satisfied), *strong condition* (must be satisfied), etc.

Decision, hypothesis and rationale. In many cases, the engineer/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 engineer needs to try another hypothesis, (s)he can perform another instance of the same process, generating a new instance of the same product. After a while (s)he is facing a collection of instances of this product, from which (s)he wants to choose the best one (according to the requirements that have to be satisfied). A justification of the decision must be provided. Hypothesis and decision justification comprise the design rationale [MYLO,92]. The decisional aspects of process modeling are particularly addressed in [POTTS,88] (through the concept of *deliberation*) and in the NATURE project [ROLL,93]. In our model, the decision is a built-in primitive process.

History. The history of a process instance is the recorded trace of the way in which its strategy has been carried out, together with the product instances involved and the rationale that has been formulated. Since a project is an instance of the highest level process, its history collects all the design activities, all the product instances and all the rationales that appeared, and will appear, in the life of the project. The history of a product instance P (also called its *design*) is the set of all the process instances, product instances and rationales which contributed to P. For instance, the design of a database collects all the information needed to describe and explain how the database came to be what it is.

4.2 Structure of a history

Depending on the complexity of the engineering process, and on the intelligence (e.g. the methodology-awareness) of the recorder, the history can be available in different formats. We will describe two dimensions according to which histories can be classified.

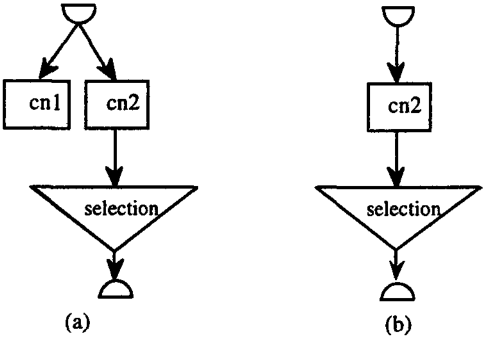


Figure 4 - The history of Fig. 3c has been restructured as a tree structure (a), and as a linear structure (b). The product instances have been hidden.

4.2.1 History topology

Technically speaking, a history can be materialized by a sort of *log file*, and therefore is a pure sequence of operations. However, if we consider multi-hypotheses approaches and decision processes, this sequence can be interpreted as a more complex graph. In general, a history has a directed acyclic graph structure, as illustrated in *Fig. 3c*. We can simplify an arbitrary history by associating every decision process instance with the successful branch, as suggested in *Fig. 4a*. In this way, a history appears as a **tree**.

Now, let us consider the successful branches only. We remove all the branches corresponding to hypotheses which have not been retained, and whose end products have been discarded. Keeping the live branches only produces a **linear** history (*Fig. 4b*). This derived history is important since it describes the way the final products could have been obtained should the engineer have proceeded without any hesitation : replaying this history on the source products will yield the same output products as the actual process did.

This classification can be generalized to methods comprising parallel branches, but this case is beyond the scope of this paper.

4.2.2 Level of aggregation

A history can be presented at different levels of detail. *Fig. 3* and *4* describe instances of C_NORM only, i.e. cn1 and cn2. In a more detailed presentation, the history of cn1 and cn2 would have been described as well, showing instances of C_TRANSFORM and C_EVALUATE. Developing in this way all the instances of engineering processes produces a complete history that can be given two extreme presentations.

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

A CASE tool must be able to present a history at various levels of aggregation.

4.2.3 Additional definitions

A history H_p is a subset of history H_n ($H_p \subseteq H_n$) if all the process instances of H_p appears in H_n, in the same order.

A history H can be sliced into sequences of process instances h_1, h_2, h_3, etc. We will note this decomposition $H = \langle h_1 \; h_2 \; h_3 \; \ldots \rangle$, where h_1 h_2 h_3 are *history slices*.

Let us consider history slice $h \subseteq H$, which starts at a time point where product instance p_1 is known to be available. h can be seen as the history of a process instance, possibly fictive, which produces product instance p_2. We can write : $p_2 = h(p_1)$.

Due to the transformational interpretation of engineering processes, a history slice can be considered as a transformation. Therefore, the structural functions are valid as well, and we can use the expressions $C_-(h)$, $C_+(h)$, $C_0(h)$.

4.2.4 Independent history slices

Let us consider history $H_0 = \langle .. \, h_1 \, .. \, h_2 \, .. \rangle$, in which we identify slices h_1 and h_2. The question addressed is : does the execution of h_2 depend on the execution of h_1, or are they independent, in which case they can be (or could have been) executed in any order, or even in parallel ? First, we define the partial order relation $\mathrm{before}(h_i, h_j)$, that states that slice h_i must be performed before h_j. This relation is defined as follows[6] :

$$\mathrm{before}(h_i, h_j) \quad \Leftrightarrow \quad C_0(h_i) \cap C_-(h_j) \neq \emptyset \vee C_+(h_i) \cap C(h_j) \neq \emptyset$$

Intuitively, h_j must follow h_i if h_j deletes *catalytic* elements of h_i, or if h_j uses constructs created by h_i. Then we define tr-before, the transitive closure of before :

[6] This relation is called the *precedence* relation in [BATINI,93].

$$\texttt{tr-before}(h_i,h_j) \Leftrightarrow \texttt{before}(h_i,h_j) \vee (\exists h \subseteq H : \texttt{tr-before}(h_i,h) \wedge \texttt{before}(h,h_j))$$

Finally, h_1 and h_2 are independent in H_0 *iff*

$$\neg\,\texttt{tr-before}(h_1,h_2) \wedge \neg\,\texttt{tr-before}(h_2,h_1)$$

4.2.5 Equivalent histories

Two histories (or history slices) H_0 and H_1 are equivalent *w.r.t.* product instance p *iff*

$$H_i(p) = H_j(p).$$

Let us consider history H_0, which is expressed as a sequence of four subsequences :

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

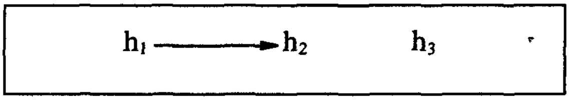
where h_1 and h_4 are (possibly empty) sequences of operations and h_2 and h_3 are two (non empty) history slices.

If we can prove that h_2 and h_3 are independent slices, then they can be swapped in H_0, leading to history $H_1 = <h_1\ h_3\ h_2\ h_4>$. Therefore, H_i is equivalent to H_j *iff* H_j can be built from H_i through a sequence of swap operations applied to independent slices.

Let us consider history H, which transforms the schema of Figure 1 into a relational schema : multivalued attribute DETAIL is transformed into entity type DETAIL and one-to-many rel-type from, then the latter and rel-type of are expressed as foreign keys.

```
H ≡ h1: (DETAIL,from) ← Att-to-ET/Value(ORDER,DETAIL)
    h2: {ORD-ID} ← RT-to-FK(from,DETAIL)
    h3: {CUST-ID} ← RT-to-FK(of,ACCOUNT)
```

The graph of tr-before is as follows :



Therefore, <h2,h3> and <h1,h3> are independent and swappable. According to the definition, <h3 , h1 , h2> and <h1 , h3 , h2> are equivalent to H, while <h3 , h2 , h1> is not equivalent.

4.2.6 Minimal history

The history of a design process records the results of the decisions, be they right or wrong, of trials, errors, backtrackings, undos and redos which shape all the exploratory human activities. Histories generally have a complex structure including several branches which materialize the exploration of concurrent hypotheses (see *Fig. 3c* for instance), of which one only led to the discovery of a target concept, the other ones being abandoned. Cycles of *doing*, then *undoing*, and finally *redoing*, are not uncommon either. Such structures must be simplified : multiple branches must be reduced to the only one that has proved useful, useless loops must be discarded. Hence the concept of **minimal history**, which can be defined as follows :

history H is minimal w.r.t. product instance p
iff for any $H' \subset H$, $H'(p) \neq H(p)$

In other words, there is no proper subsets of H which still are equivalent to H. Given history H, H_m is a *minimal version* of H if H_m is minimal, and H_m is equivalent to H.

5. Database reverse engineering

Both reverse engineering and design recovery must be defined against a reference forward engineering approach. Indeed, reverse engineering can be modeled as the reverse of designing a database, and design recovery consists in developing a plausible history of a definite design methodology.

5.1. The reference database design methodology

In this section, we draw the main principles of standard database design methodologies as described in [TEOREY,94] and [BATINI,92] for instance. Traditionally, database design can be described as made up of a sequence of four specific processes, namely CONCEPTUAL DESIGN, LOGICAL DESIGN, PHYSICAL DESIGN and VIEW DESIGN.

- The CONCEPTUAL DESIGN process is aimed at producing the *conceptual schema,* a computer-independent representation of the information to be stored into the future database. Among others, this representation is to be given such desirable qualities as normality, minimality, clarity, that can be considered as provided through a specific sub-process called CONCEPTUAL NORMALIZATION.
- Through the LOGICAL DESIGN process, the conceptual schema is transformed into a *DMS-compliant optimized logical schema* with the following three characteristics :
  - *equivalence* : it expresses the same semantics as the conceptual schema,
  - *DMS-compliance* : it follows the data model of the chosen DMS[7],
  - *efficiency* : it satisfies operational or technical criteria such as space and time performance.

  This process can be considered as being refined, at least conceptually, into three subprocesses : schema simplification, schema optimization and DMS-translation (*Fig. 5*).

  - SCHEMA SIMPLIFICATION replaces *advanced constructs* such as IS-A hierarchies, N-ary rel-types by equivalent basic constructs. The simplified conceptual schema is perceived as an adequate medium for logical reasonings when traditional DMS are considered.
  - SCHEMA OPTIMIZATION modifies the schema in order to give it better performances.
  - SCHEMA TRANSLATION converts the schema into data structures that are compliant with the model of the DMS. The result consists into two complementary parts : the DMS constructs, that can be controlled by the DMS, and the non-DMS constructs, most often integrity constraints, that will be ignored by the DMS.

[7] A DMS is a Data Management System. It is either a File Management System, or a Database Management System.

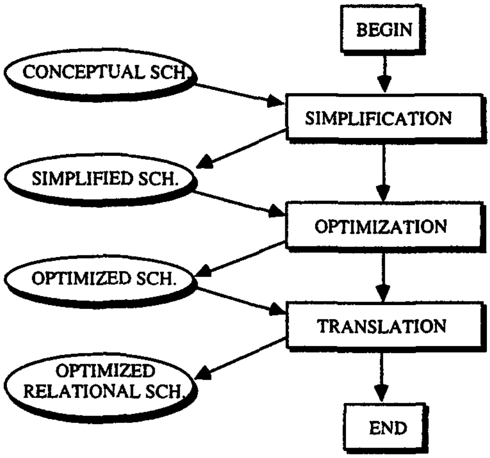


Figure 5 - Development of the Logical Design process of the reference design methodology.

- The PHYSICAL DESIGN process translates the DMS constructs of this schema into a DMS-DDL text, and the non-DMS constructs into, e.g., procedural sections or local variables of the application programs. In addition, through physical tuning, technical parameters are set and physical constructs are built, such as indexes and clusters.
- The VIEW DESIGN process builds the external views required by users and programs, and translates them in DMS-DDL and application programs fragments in the same way as for the global schema.

Most of these processes can be in turn refined, directly or indirectly, into lower-level processes or operators that are primitive schema transformations as defined in *Section 3*.

5.2 Database reverse engineering

Grossly speaking, this process transforms input products, that mainly consist of source code texts, into schemas of the database. Two schemas are of particular interest, namely the *logical schema of the database* according to the DMS model, and a possible *conceptual schema*. The main input products consist in DMS-DDL description of the global schema and of the views, either in text format, or as data dictionary contents. Essential information, for instance on untranslated integrity constraints, can be found in the source text of the application programs (DML queries, procedure and data structures), in screen layout and procedural components of the user interface and in database checking procedures (e.g. triggers, check clauses). The physical schema may yield useful hints concerning the logical schema (indexes suggesting foreign keys, join-based clusters, etc). File/database contents analysis may provide strong hints, or validate hypotheses, on the presence of such constructs as identifiers, foreign keys, field layout and value domains.

In [HAI,93a] and [HAI,95a], a general procedure is proposed for reverse engineering any database or collection of files. It consists into two major processes, namely DATA STRUCTURE EXTRACTION and DATA STRUCTURE CONCEPTUALIZATION (*Fig. 6*).

- DATA STRUCTURE EXTRACTION produces a complete description of the data structures according to the model of the DMS, e.g. COBOL file structures, CODASYL schema, relational schema, etc. In addition, the non-DMS parts of the schema have been elicited from, a.o. the procedural parts of the applications or database contents. According to the DMS, the process can be more or less easy. For instance, the DMS part of COBOL data structures can be difficult to discover while CODASYL or relational schemas can be analysed more easily. DATA STRUCTURE EXTRACTION appears as the inverse of the PHYSICAL DESIGN forward process.

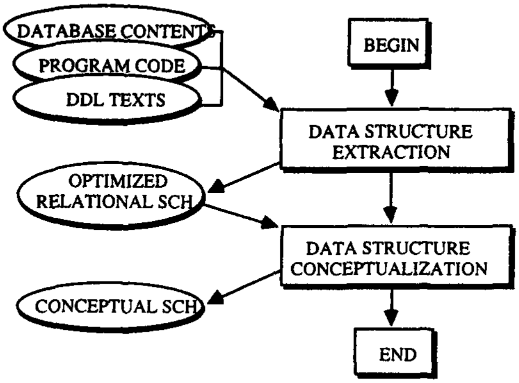


Figure 6 - Database Reverse Engineering : the major processes.

- DATA STRUCTURE CONCEPTUALIZATION tries to make the semantics of the logical schema explicit by recovering the intention of the optimized DMS data structures. This process is to a large extent the reverse of the LOGICAL DESIGN forward process, and can be decomposed into three subprocesses (*Fig. 7*).

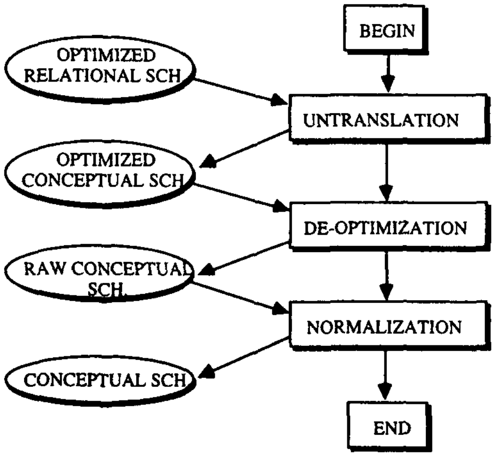


Figure 7 - Database Reverse Engineering : Development of the Data Structure Conceptualization process.

- UNTRANSLATION detects DMS-compliant constructs and replaces them by their DMS-independent equivalent. This process appears as the inverse of the SCHEMA TRANSLATION forward process.

- DE-OPTIMIZATION detects and removes the non-semantic constructs from the logical schema, and particularly the optimization structures. This process appears as the inverse of the SCHEMA OPTIMIZATION forward process.
- CONCEPTUAL NORMALIZATION has the same objectives as its forward engineering counterpart such as minimality, clarity and corporate standard compliance. In particular, it is intended to recover the high-level structures transformed by the SCHEMA SIMPLIFICATION.

According to this approach, each DBRE process is either a forward process or the inverse of a forward process. Consequently, they can be based on basic schema transformations that are either forward transformations or the inverse thereof.

6. Database design recovery

In this section, we suppose that we have been provided with the history of a reverse engineering process which has led to a correct, coherent and validated conceptual schema of a legacy system. We will shortly describe a general procedure to built in a systematic way a possible design history of this system.

6.1 Normalizing the DBRE history

The objective is to produce a correct and minimal DBRE history which complies with the standard way this process should have been carried out. The normalization includes two processes, namely minimizing and restructuring the history.

A **minimal** history is first extracted. Indeed, database reverse engineering basically is an exploratory process, and quite naturally, its history will include useless loops and dead branches. Getting rid of these structures produces a minimal history, as defined in *Section 4.2.6*. We give some detail about this minimization process.

- Removing dead branches provides a history in which only the branches and products that contribute to the final product (the conceptual schema) are kept. It consists in parsing the history backward, from the final product toward the input products, and marking the process instances and product instances examined. The unmarked instances are discarded. This process is fairly easy and can be automated.
- Detecting and reducing useless sequences, and particularly useless loops are more complex problems. Though the problem has not been completely formalized yet, we can propose the following heuristics :

  we consider history H, which is expressed as a sequence of transformations :

  $$H = <T_1\ T_2\ T_3\ T_4>,$$

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

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

  In short, we can remove any pair of transformations which prove to be the inverse of each other, and whose target objects are not used in any other transformations. This heuristics is still valid when the T's are history slices instead.

Then this history is **restructured** according to the reference RE methodology. In particular, the operations are swapped, provided they are independent, and reclassified into the reference engineering process instances in which they best fit, and the sequences are sorted in order to produce clusters of similar operations. These manipulations must preserve the equivalence of the source and final histories, according to the definitions in *Section 4.2.*

6.2 Replacing each operation with its inverse

Each operation of the normalized history is replaced by its inverse. Two cases will occur. The first one concerns primitive process instances. For each of them, there exists at least one inverse operation (e.g. *transforming an entity type into an attribute* has at least two inverse transformations, as illustrated in *Section 3.6.3*). The most adequate among them is chosen.

The second case regards the engineering process instances. According to the way in which the reference DBRE methodology has been designed (*Section 5.2*), most of its processes are, at least partially, the inverse of a forward process. Therefore, each reverse process instance is replaced, possibly after some redistribution of its contents, by its inverse forward process instance. Let us consider some representative examples :

parsing	→	code generation
conceptualization	→	logical design
normalization	→	simplification
untranslation	→	translation
un-optimization	→	optimization
remove redundancies	→	introduce redundancies

6.3 Reversing the order of the operations

In each engineering process instance, whatever its level of aggregation, the order of the operations is reversed.

6.4 Normalizing the design history

The result of the preceeding processes is a correct design history. If the DBRE history has been correctly normalized, the resulting design history should enjoy the same minimality properties. However, when compared with the history of a native design project, it will generally appear as somewhat ackward. The objective of this normalization is to produce a more natural design history according to the corporate methodology standards, i.e. the *reference methodology.* This will be obtained by reordering, sorting and grouping the operations in each engineering process instance. For instance more than one instance of the optimization process can appear in the history. They should be merged in order to group them in one contiguous slice. In addition, several similar operations in the same engineering process instance can be grouped if they have the same objective. For instance, all the *transform-rel-type-into-foreign-key* operations are packaged into one contiguous sequence.

Informal or linguistic information is associated with the important objects of the history. For instance, each data object must be given a semantic description, and for each important process instance the rationale must be documented [BIGG,89]. Recovering the rationale of the decisions is a hazardous task. However, it can be improved by the context provided by the reference methodology. In each engineering process of this methodology, several ways of working are generally proposed, each driven by specific design criteria, such as space optimization, time optimization, normality, etc. By comparing the history profile with each of these suggested way of working, we can more easily guess what were the design criteria the developer had in mind when (s)he built the legacy system. For instance, transforming a multivalued attribute into serial attributes (3.6.4) suggests access time minimization, while transforming it into an entity type (3.6.3) suggests that readability and maintainability have been given higher priority.

6.5 Generalizing the design history

As it is described so far, a history traces elementary transformations only, be they primitive or enginering process instances. Indeed, each operation specifies the application of a generic transformation to an individual specification object : *transform attribute PUBLISHER into an entity type*, *transform entity type BUYS into a relationship type*, and so on. This has some important consequences, of which we will mention two.

1. Such a detailled history does not help much in understanding the reasonings underlying the (hypothetical) design of the system, in the same way as reading the log of the elementary manipulations performed by a surgeon during an operation does not inform much a medical student on the objectives and on the strategies used.

2. Replaying this history works fine on the recovered abstract specifications, since it yields exactly the source logical schema, but it can lead to unsatisfying results when applied on modified abstract specifications, a scenario that is common in database maintenance and evolution.

The problem can be stated as *trying to recover the specific strategy used to perform each process*. This problem has been tackled in the software engineering domain, where researches have been conducted in recovering a possible program from traces of its execution.

These problems can be solved, at least partially, by **generalizing** the history, i.e. by replacing some sequences of similar individual operations by higher-level rules of which these sequences are the trace. For example, let us suppose that the schema on which engineering process instance I is performed includes relationship types R1, R2 and R3 which all satisfy predicate P (e.g. is_N-ary), and that no other objects satisfy P. Let us also suppose that, in I, transformation RT-to-ET is applied on R1, R2 and R3 :

```
E1 ← RT-to-ET(R1)
E2 ← RT-to-ET(R2)
E3 ← RT-to-ET(R3)
```

Technically speaking, we can replace this sequence with the *predicate-driven global transformation* :

for each rel-type R_i such that is_N-ary(R_i) do : E_i ← RT-to-ET(R_i)

When generalized in this way, the history is much shorter, it provides a clearer and more explicit specification of the reasonings driving the process instance (e.g. the intention of the origin history fragment obviously was to *transform all N-ary rel-types into entity types*), and it makes the history applicable to modified versions of the abstract specifications. For instance, let us consider that, due to changes in the user requirements, a new N-ary relationship type R4 has to be added to the conceptual schema. Replaying the first version of the history of process I will leave R4 unchanged, while applying the generalized version will process R4 as well, therefore complying with the intention of I.

As for most inductive reasonings, this process cannot be fully automated. Indeed, only a skilled designer can validate such a generalization. However, detecting similar transformation patterns, grouping them, and suggesting candidate generalizations can be described by formal rules, and supported by knowledge-based CASE functions.

Once again, generalization is a complex process that can be made much easier and more deterministic if a precise reference design methodology is available. Indeed, each way of working proposed to solve the problem addressed in the current process should be supported by some standard transformations plans. The problem now reduces to identifying the most appropriate plan, and trying to map sequences of individual transformations to predicate-driven global transformation.

7. CASE support of design recovery

Histories are the baseline of design recovery. Recording, managing, and processing engineering histories requires strong CASE support. From the discussion developed in this paper we can lay down the main requirements such tools should satisfy. We will mention some of them, then describe shortly a prototype CASE tool intended to support history management and design recovery.

7.1 Requirements for CASE support of design recovery

An environment that is to assist developers in design recovery of database should include some important functions.

1. The repository of the tool must accomodate specification products at various levels of abstraction, and according to various common representation paradigms.
2. Semantics-preserving transformations are the building blocks of history formalisation. Therefore, the tool must support some kind of transformational approach.
3. The tool must provide the basic functions for reverse engineering.
4. The tool must be methodology-independent : it will allow various engineering strategies, ranging from disciplined and formal approaches to informal, pragmatic and experimental ones.
5. However, the tool must be methodology-aware, and include an active method model. It must also include a method specification language, a compiler for this language, and a method engine which provide method enactment to ensure that the actions performed by the developers comply with the current method
6. The method engine must include a history recorder coupled with the current method.
7. It must include various functions for history management : viewing at different aggregation levels, restructuring, reversing, pruning, analysing, updating, annotating.
8. Finally, it must include a replay processor.

7.2 The DB-MAIN CASE tool

DB-MAIN is both a toolset to support system engineering, and a CASE tool development environment (i.e. a *Meta-CASE* or a *CASE tool factory*).It is one of the main products of the DB-MAIN project, dedicated to database applications engineering, and more particularly to the problems that arise when such applications evolve and have to be maintained. Its basic principles and its architecture have been described in [HAI,95b] and [HAI,96b]. DB-MAIN Version 1.0 offers both basic and sophisticated services for database forward and reverse engineering. Though coping with the very problem of database application evolution and maintenance in a significant way is due to 1997, the current version, and the current developments comprise important aspects that are to satisfy the requirements mentioned above.

In its first version (October 1995), the tool offers support for most forward and reverse engineering activities. More specifically, it includes the following functions and components : specifications management; representation of all the project products through a generic, wide-spectrum, representation model for conceptual, logical and physical objects; multiple views of the specifications (4 hypertexts and 2 graphical views); a toolbox of more than 25 semantics-preserving transformational operators, allowing to carry out in a systematic way such activities as conceptual normalization, or the development of optimized logical and physical schemas from conceptual schemas, and conversely (i.e. reverse engineering); code generators and parsers; interactive and programmable source text and name analysers and processors.

It also includes a history manager which records the engineering activities of the analyst, and which allows their further automatic or assisted replay; this manager can also reverse an existing history. A series of assistants (problem-driven expert modules) have been developed to assist the analyst in frequent, tedious, knowledge-based or complex tasks.

The tool also allows the development of *specialized CASE tool*, either according to specific methodologies, or by adding new concepts and new functions to the basic toolset.

The current version of the history manager, despite its early prototype state, already provides us with an experimental workbench to study the assisted building of forward histories from reverse histories. Developing an improved history manager is one of the objectives of Version 2 of DB-MAIN.

DB-MAIN have been developed in C++ for PC/Windows workstations. It is currently used by several industrial partners, mainly in reverse engineering, redocumentation, reengineering and migration projects. An education/evaluation version[8] is used in an increasing number of schools and universities, and can be obtained free of charge by non-profit organizations.

8. A short case study

Let us consider the relational schema in *Fig. 8*, which could be the result of carrying out the *Data Structure Extraction* process on the DDL and procedural texts of an application.

Interpreting this schema, i.e. rebuilding a possible conceptual schema, is the job of the *Data Structure Conceptualization* process. This process comprises three main subprocesses : *Untranslation*, *De-optimization* and *Normalization*. The first two ones can be conducted in parallel, while the third one will generally be carried out as the finishing touch. Let us imagine the following scenario, immediately translated into its history :

- we express the first foreign key, RNAME, as the relationship type `from`; we start the history with an *Untranslation* action :

  A1: from ← FK-to-RT(CUSTOMER, {RNAME}, REGION)

[8] Fully functional, but limited to 1000-object projects.

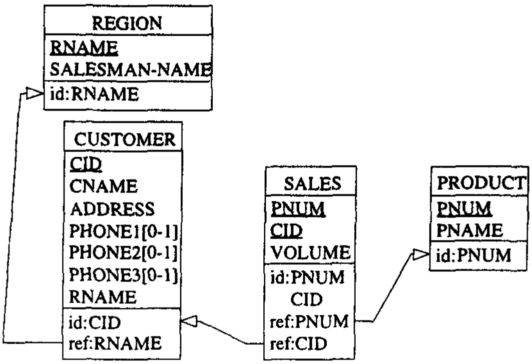


Figure 8 - The logical schema of a relational database

- we then observe the structure of serial attributes {PHONE1, PHONE2, PHONE3} in the CUSTOMER table; we know it as a common low cost implementation of a multivalued attribute; we express these attributes as the multivalued attribute PHONE :

  A2: PHONE ← SerialAtt-to-MultiAtt(CUSTOMER, {PHONE1, PHONE2, PHONE3})

  This action obviously relates to the *De-optimization* process.

- the schema includes two other foreign keys, which are known to express many-to-one relationship types; we augment the history with two *Untranslation* actions :

  A3: r1 ← FK-to-RT(SALES, {CID}, CUSTOMER)

  A4: r2 ← FK-to-RT(SALES, {PNUM}, PRODUCT)

- the attribute SALESMAN of REGION is perceived as a low-level representation of an important concept, salesmen, that will feel better as an entity type. This attribute is transformed into an entity type :

  A5: (SALESMAN, in) ← Att-to-ET/Value(REGION, SALESMAN-NAME)

  The representation seems clearer and more readable. This is a *Normalization* transformation.

- the SALES entity type is perceived as the representation of a relationship type. This decision belongs to the *Normalization* process :

  A6: SALES ← ET-to-RT(SALES)

- finally, the idea to express salesmen as an entity type is not a so good one, because it makes the schema more complex without any profit. We choose to transform this entity type into an attribute. This still is a *Normalization* operation :

  A7: SALESMAN-NAME ← ET-to-Att(SALESMAN)

The actions are classified according to the process to which they are most relevant. The history of the conceptualization process of this reverse engineering project is summarized as follows.

Untranslation

A1: from ← FK-to-RT(CUSTOMER, {RNAME}, REGION)

A3: r1← FK-to-RT(SALES, {CID}, CUSTOMER)

A4: r2 ← FK-to-RT(SALES, {PNUM}, PRODUCT)

De-optimization

A2: PHONE ← SerialAtt-to-MultiAtt(CUSTOMER, {PHONE1, PHONE2, PHONE3})

Normalization

A5: (SALESMAN, in) ← Att-to-ET/Value(REGION, SALESMAN-NAME)

A6: SALES ← ET-to-RT(SALES)

A7: SALESMAN-NAME ← ET-to-Att(SALESMAN)

A graphical view of this history is proposed in *Fig. 9*. The instances of the primitive processes are represented by their labels.

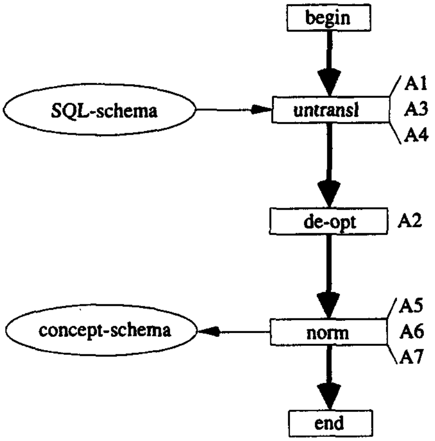


Figure 9 - The history of the reverse engineering of the relational schema of Fig. 8

The resulting conceptual schema appears in *Fig. 10*.

Building a possible design history

Let us consider the standard database design methodology whose *Logical design* phase is described in *Fig. 5*. It is obvious that following this strategy, it should be possible to produce a large variety of equivalent logical relational schemas. One of them is the source schema of *Fig. 8*. The objective is to build the unique history that produces exactly this logical schema from the recovered conceptual schema. As proposed in *Section 6*, we apply the 5-step strategy :

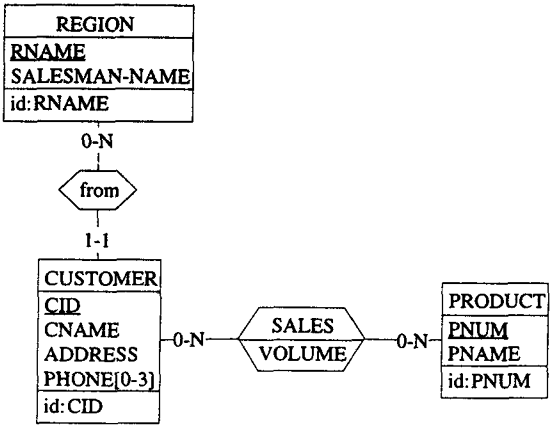


Figure 10 - A possible conceptual schema for the relational schema of Fig. 8.

1. *Normalizing the DBRE history*

Despite its simplicity, this history need be normalized. It appears that operations A5 and A7 form a useless loop. Indeed, we have :

- Att-to-ET/Value = ET-to-Att^{-1}
- C_-(A7) = {SALESMAN, SALESMAN-NAME, in, in.REGION, in.SALESMAN, id(SALESMAN)} = C_+(A5)
- no transformations use C_+(A5) = {SALESMAN, SALESMAN-NAME, in, in.REGION, in.SALESMAN, id(SALESMAN)}, but A7
- no transformations use C_+(A7) = {SALESMAN-NAME}

Therefore A5 and A7 can be removed.

2. *Replacing each operation with its inverse*

Both engineering and primitive operations have direct inverse :

Translation

{RNAME} ← RT-to-FK(from,CUSTOMER)

{CID} ← RT-to-FK(r1,SALES)

{PNUM} ← RT-to-FK(r2,SALES)

Optimization

{PHONE1,PHONE2,PHONE3} ← MultiAtt-to-SerialAtt(CUSTOMER,PHONE)

Simplification

(SALES,{(CUSTOMER,r1),(PRODUCT,r2)}) ← RT-to-ET(SALES)

3. Reversing the order of the operations

Simplification

```
(SALES, {(CUSTOMER,r1), (PRODUCT,r2)}) ← RT-to-ET(SALES)
```

Optimization

```
{PHONE1, PHONE2, PHONE3} ← MultiAtt-to-SerialAtt(CUSTOMER, PHONE)
```

Translation

```
{PNUM} ← RT-to-FK(r2, SALES)
{CID} ← RT-to-FK(r1, SALES)
{RNAME} ← RT-to-FK(from, CUSTOMER)
```

4. Normalizing the design history

Due to its simplicity, this history need not be normalized.

5. Generalizing the logical design history

The **Simplification** process includes one operation only. However, it is an instance of a standard way to get rid of N-ary relationship types, a typical complex construct. Therefore it is reasonable to generalize this instance into the *transformation of all the N-ary relationship types into entity types.*

The **Translation** process includes a sequence of similar operations which are good candidates for generalization : *express all functional relationship types into foreign keys.*

The **Optimization** process is much more difficult to generalize. Indeed, it often relies on complex reasonings strongly linked with the developer's skill, the technology state at development time, and with the data statistics and of the application requirements at the same time. These conditions may have changed since then. For these reasons, and in the absence of additional information, we propose not to generalize the operation(s).

The result is presented in Fig. 11.

9. Conclusion

We have shown in this presentation that database design recovery can be tackled through reverse engineering techniques which are now fairly well understood and mastered. We have also shown that the proposed approach can be formalized thanks to the concept of semantics-preserving transformation. Moreover, this formality provides a sound basis for CASE support, as illustrated by the DB-MAIN CASE tool, where prototype functions of history recording and management have been implemented. It must be admitted, however, that design recovery, as interpreted by [BIGG,89], generally goes beyond purely formal treatment. In particular, explaining why some decisions have been taken, and declaring how the reconstructed artifacts relate with the application domain is up to the analyst, who is supposed to elicit such reasonings and relations from the technical traces which (s)he faces during reverse engineering activities.

Logical design

Simplification

for each rel-type R such that (R = R(E1,..,Ei) & ¬ (R is functional)) do :

```
        (R,{(E1,r1),..,(Ei,ri)}) ← RT-to-ET(R)
```

Optimization

```
{PHONE1,PHONE2,PHONE3} ← MultiAtt-to-SerialAtt(CUSTOMER,PHONE)
```

Translation

for each rel-type R such that (R = R(E1,E2) & id(E1) = A) do :

```
        A ← RT-to-FK(R,E2)
```

Figure 11 - A possible design history for the relational schema of Fig. 8.

An immediate question arises when a correct, concise and documented design is available: what to do with it ? For instance, how can we use this product to carry out system maintenance and evolution in a more systematic way than we now do ? This problem is beyond the scope of this paper, but a partial answer can be found in [HAI,94].

The data structure domain is fairly well formalized, and has long led to models and tools which are both formal and intuitive (and therefore used by practitioners). The domain of software engineering is much larger, and far less formalized, at least as far as practice is concerned. Hence a second question which can be asked at this stage : can these principles be generalized to help recover the design of other aspects of a legacy system, and in particular of the programs ? A tentative positive answer has been developed in [HAI,95a] for reverse engineering, but the question is open as far as history recovery is concerned.

9. References

[BALZER,81] Balzer, R., Transformational implementation : An example, *IEEE TSE*, Vol. SE-7, No. 1, 1981

[BATINI,92] Batini, C., Ceri, S., Navathe, S., B., *Conceptual Database Design*, Benjamin/Cummings, 1992

[BATINI,93] Batini, C., Di Battista, G., Santucci, G., Structuring Primitives for a Dictionary of Entity Relationship Data Schemas, *IEEE TSE*, Vol. 19, No. 4, 1993

[BIGG,89] Biggerstaff, T., J., Design Recovery for Maintenance and Reuse, *IEEE Computer*, July 1989

[FICKAS,85] Fikas, S., F., Automating the transformational development of software, *IEEE TSE*, Vol. SE-11, pp1268-1277, 1985

[HAI,91] Hainaut, J-L, Database Reverse Engineering, Models, Techniques and Strategies, in *Proc. of the 10th Conf. on Entity-Relationship Approach*, San Mateo (CA), E-R Institute, 1991

[HAI,92] 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), Manchester, May 1992, Springer-Verlag, LNCS, 1992

[HAI,93a] 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

[HAI,93b] 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, 1993

[HAI,94] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D., Evolution of database Applications : the DB-MAIN Approach, in *Proc. of the 13th Int. Conf. on ER Approach*, Manchester, Springer-Verlag, 1994

[HAI,95a] Hainaut, J-L, *Database Reverse Engineering - Problems, Techniques and Tools*,Tutorial notes, CAiSE'95, Jyväskylä, Finland, June 1995 (available at jlh@info.fundp.ac.be)

[HAI,95b] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D., Requirements for Information System Reverse Engineering Support, in *Proc. of the 2nd IEEE WC on Reverse Engineering*, Toronto, July 1995, IEEE Computer Society Press, 1995.

[HAI,95c] Hainaut, J-L, *Transformation-based database engineering*,Tutorial notes, VLDB'95, Zürich, Switzerland, Sept. 1995 (available at jlh@info.fundp.ac.be)

[HAI,96] Hainaut, J-L, Specification preservation in schema transformations - application to semantics and statistics, *Data & Knowledge Engineering*, Vol. 11, No. 1, 1996.

[HAI,96b] Hainaut, J-L, Englebert, V., Henrard, J., Hick J-M., Roland, D., Database Reverse Engineering : from Requirements to CASE tools, *Journal of Automated Software Engineering*, Vol. 3, No. 1, 1996

[HALPIN,95] Halpin, T., A., Proper, H., A., Database schema transformation and optimization, in *Proc. of the 14th Int. Conf. on ER/OO Modelling (ERA)*, Dec. 1995

[KOBA,86] Kobayashi, I., Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence, in *Information Systems*, Vol. 11, No 1, pp. 41-59, January, 1986

[KOZA,87] Kozaczynsky, Lilien, An extended Entity-Relationship (E2R) database specification and its automatic verification and transformation, in *Proc. of Entity-Relationship Approach*, 1987

[MYLO, 92] Mylopoulos, J., Chung, L., Nixon, B., Representing and Using Nonfunctional requirements : A Process-Oriented Approach, *IEEE TSE*, Vol. 18, No. 6, June 1992

[NAVA,80] Navathe, S., B., Schema Analysis for Database Restructuring, in *ACM TODS*, Vol.5, No.2, June 1980

[POTTS,88] Potts, C., Bruns, G., Recording the Reasons for Design Decisions, in *Proc. of ICSE*, IEEE, 1988

[RAUH,95] Rauh, O., Stickel, E., Standard Transformations for the Normalization of ER Schemata, in *Proc. of the CAiSE•95 Conf.*, Jyväskylä, Finland, LNCS, Springer-Verlag, 1995

[ROLL,93] 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)

[ROSE,88] Rosenthal, A., Reiner, D., Theoretically sound transformations for practical database design, in *Proc. of Entity-Relationship Approach*, 1988

[ROSE,94] Rosenthal, A., Reiner, D., Tools and Transformations - Rigourous and Otherwise - for Practical Database Design, *ACM TODS*, Vol. 19, No. 2, June 1994

[SPACCA,92] Spaccapietra, S., Parent, C., View Integration : A Step Forward in Solving Structural Conflicts, *IEEE Trans. on Knowledge and Data Engineering*, October, 1992

[TEOREY,94] Teorey, T. J., *Database Modeling and Design : the Fundamental Principles*, Morgan Kaufmann, 1994