in *Generative and Transformational Software Engineering*, LNCS, Springer, 2006

The Transformational Approach to Database Engineering

Jean-Luc Hainaut[1]

[1] University of Namur, Institut d'Informatique
Rue Grandgagnage, 21 B-5000 Namur, Belgium
jlh@info.fundp.ac.be
http://www.info.fundp.ac.be/libd

Abstract. In the database engineering realm, the merits of transformational approaches, that can produce in a systematic way correct, compilable and efficient database structures from abstract models, has long be recognized. Transformations that are proved to preserve the correctness of the source specifications have been proposed in virtually all the activities related to data structure engineering: schema normalization, logical design, schema integration, view derivation, schema equivalence, data conversion, reverse engineering, schema optimization, wrapper generation and others. This paper addresses both fundamental and practical aspects of database transformation techniques. The concept of transformation is developed, together with its properties of semantics-preservation (or reversibility). Major database engineering activities are redefined in terms of transformation techniques, and the impact on CASE technology is discussed. These principles are applied to database logical design and database reverse engineering. They are illustrated by the use of DB-MAIN, a programmable CASE environment that provides a large transformational toolkit.

1 Introduction

Data structure manipulation has long proved to be a fertile domain for transformational engineering process modelling. Several contributions have made this approach a fruitful baseline to solve the complex mapping problems that are at the core of many database engineering processes.

We can mention the normalization theory, which laid the basis for data- and constraint-preserving schema transformations [13], but also the now standard 3-schema data modeling architecture [48] which clearly complied, more than 25 years ago, to what the SE community currently calls *Model-Driven Engineering* (MDE). Generally built on these principles, most database design methodologies rely on four expressions of the database structure, namely the conceptual schema, the logical schema, the physical schema and the DDL[1] code (Fig. 17). According to these approaches, a schema at one level derives from a more abstract schema at the upper level through

[1] Data Description Language. That part of the DBMS language dedicated to the creation of data structures.

some kind of translation rules that preserve its information contents, which clearly are schema transformations. For instance, a logical relational schema can be produced from the conceptual schema by applying to non SQL-compliant conceptual structures rewriting rules that produce relational constructs such as tables, columns and keys. If the rules are carefully selected, the relational schema has the same information contents as its conceptual origin.

An increasing number of bodies (e.g., the OMG) and of authors recognize the merits of transformational approaches, that can produce in a systematic way correct, compilable and efficient database structures from abstract models.

Transformations that are proved to preserve the correctness of the source specifications have been proposed in virtually all the activities related to schema engineering: schema normalization [39], logical design [4, 19, 41], schema integration [4, 34], view derivation [35, 33], schema equivalence [11, 28, 29, 32], data conversion [36, 12, 46], reverse engineering [6, 8, 18, 19], database interoperability [34, 45], schema optimization [19, 25], wrapper generation [45] and others.

Warning
In the database community, a general formalism in which database specifications can be built is called a *model*. The specification of a definite database structure expressed in such a model is called a *schema*. Example: the *conceptuel schema* of the Customer database is expressed in the *Entity-relationship model*, while its *logical schema*, that is made up of table, column and key definitions, complies with the *relational model*.

A First Illustration

Before discussing in deeper detail the concept of transformation and its properties, let us have a look at a first practical application of the concept. The schemas of Fig. 1 show a popular example, namely the production of a relational schema (top right), from a small conceptual schema (top left) that describes a set of books for which a collection of copies are available. The graphical conventions will be described later, but the essence of the schemas can be grasped without further explanation.

The main stream of the process is covered by the two top schemas. The translation rules that have been applied can be identified easily:

1. each entity type is represented by a table,
2. each single-valued attribute is represented by a column,
3. each *all-attribute* identifier is represented by a primary or alternate key,
4. each one-to-many relationship type is represented by a foreign key,
5. each multivalued attribute is represented by a table, comprising the source attribute that is declared a primary key, and by an additional table made up of a foreign key to the table that represents the entity type of the attribute and another foreign key to the new attribute table; both foreign keys form the primary key of their table.

Of course, other, more or less sophisticated, sets of rules exist, but this one is adequate for demonstration purpose.

We can read this derivation process from another, *transformational*, point of view. We do not produce another schema, but we *progressively modify* the source conceptual schema, until it complies with the structural patterns allowed by the relational model.

This interpretation, which will prove much more powerful and flexible than the translation rules approach, is illustrated in the alternate circuit (top → down → right → up) of Fig. 1.

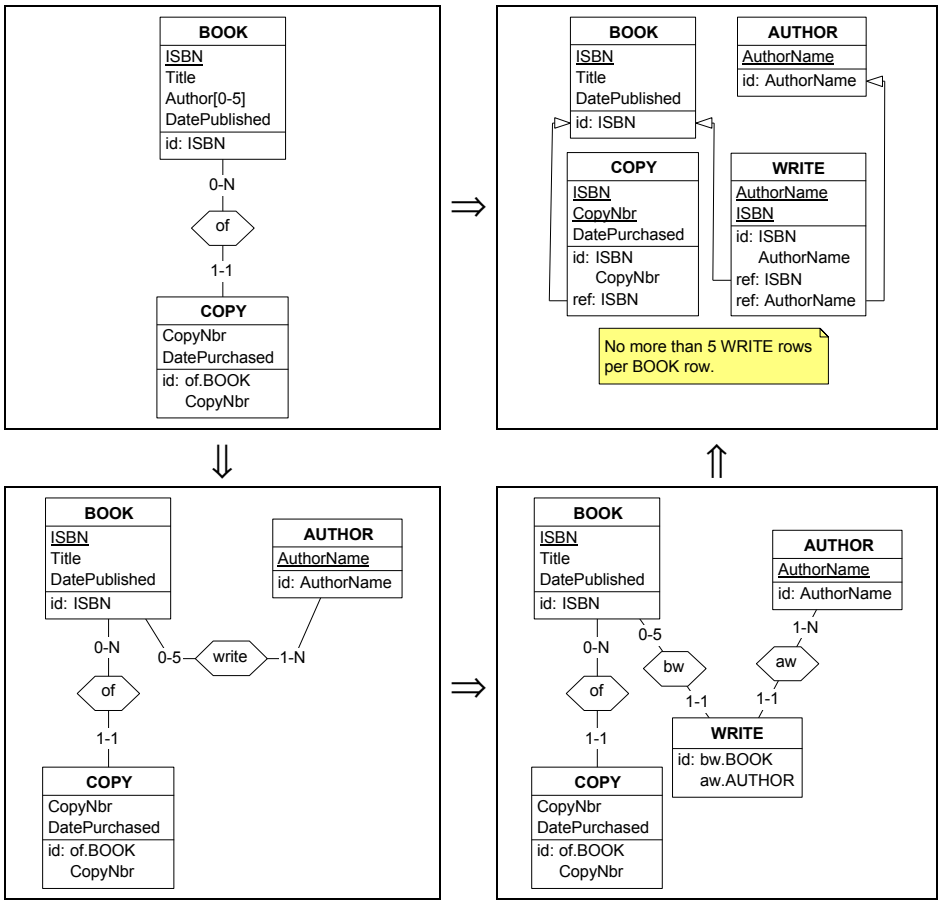


Fig. 1. Two ways to describe the derivation of a relational schema from a conceptual schema

The first modified schema (bottom left) derives from the source conceptual schema (top left) as follows: the multivalued attribute Author has been **replaced** with the entity type AUTHOR comprising the identifying attribute AuthorName, and the many-to-many relationship type write.

Then (bottom right), the new many-to-many relationship type write is **replaced** with entity type WRITE together with two one-to-many relationship types bw and aw.

The schema does not include multivalued attributes or complex relationship types anymore.

Finally, each one-to-many relationship type is **replaced** with a foreign key. Hence the final version, at the top right side.

The Structure of this Paper

This short illustration raises several questions and problems, to some of which this paper will try to answer, at least partially. The paper is organized in two parts, that allow two levels of reading.

The **first part**, that includes Sections 2 to 8, develops **practical aspects** of the transformational paradigm. Section 2 positions the role of transformation in the database realm. In Section 3, we show that dealing with multiple databases leads us to introduce a generic pivot model, the GER, that is intended to represent a large variety of operational models. Its graphical representation is sketched and a formal semantics is suggested. In this section, we also show how specific operational models can be defined in the GER. The concept of schema transformation is precisely defined in Section 4, in which the property of semantics-preservation is defined and analyzed. In Section 5, we describe some useful elementary and complex GER transformations, that are then used in Section 6 to revisit the Database Design process, showing that it is intrinsically a (complex) schema transformation. Similarly, Section 7 studies the Reverse Engineering process as an application of the transformational paradigm. Section 8 discusses the role of transformations in CASE tools, and illustrates this point with the toolkit and the assistants of DB-MAIN.

The **second part**, comprising Sections 9 to 12, develops **formal aspects** of transformations that were only sketched and suggested in Part 1. Section 9 describes the ERM, an extended N1NF[2] relational model the semantics of which is borrowed from the relational theory. Section 10 maps the GER onto the ERM so that the former can be given a precise formal semantics. Section 11 described a small set of ERM transformations that can be proved to be semantics-preserving. Finally, Section 12 exploits the GER-ERM mapping to prove the semantics-preservation property of selected practical GER transformations.

Section 13 concludes the paper.

---

[2] *Non 1st Normal Form*. Qualifies a relational structure that uses non simple domains. Elements of a non simple domain can be tuples and/or sets. In particular, a relation or the powerset of a relation can be a valid domain. A N1NF relational model is a relational model in which non simple domains are allowed.

in *Generative and Transformational Software Engineering*, LNCS, Springer, 2006

Part 1. Transformations for Database Engineering

2 Transformational Engineering

Producing efficient software by applying systematic transformations to abstract specifications has been one of the most mythical goals of software engineering since the late seventies. For instance, [3] and [14] consider that *the process of developing a program [can be] formalized as a set of correctness-preserving transformations [...] aimed to compilable and efficient program production*. In this context, according to [37], *a transformation is a relation between two program schemes P and P' (a program scheme is the [parameterized] representation of a class of related programs; a program of this class is obtained by instantiating the scheme parameters). It is said to be correct if a certain semantic relation holds between P and P'*. The revival of this dream has now got the name of Model-Driven Architecture [38], or, more generally, Model-Driven Engineering (MDA/MDE).

It is not surprising that this view has been adopted and developed by the database community since the seventies. Indeed, the data domain has relied on strong theories that can cope with most of the essential aspects of database engineering, from clean data structuring (including normalization) to operational data structures generation.

In particular, producing a target schema from a source schema can be modeled either by a set of translation rules, or by a chain of restructuring operators or transformations. The latter has proved particularly attractive, notably in complex, incremental, processes.

The question of how many operators are needed to cover the current needs in database engineering is still open, though it has been posed for long: in the 80's, authors suggested that four [15] [29] to six [11] were enough, but experience has shown that there is no clear answer, except that surely more transformations are needed, as we will show in the following.

One of the peculiarities of transformational approaches in the database realm is that they must, in all generality, cope with three aspects of the application system, namely the *data structures*, the *data*, and the *programs*. Let us consider a scenario in which a database must be migrated from a technology to another one. Clearly, this database must be *transformed*, whatever the meaning of this term, into another database. This means that three components of the application must be modified.

1. The database schema, that must comply with the data model of the target technology, and, possibly, include additional requirements that have emerged since the last schema modification.
2. The data themselves, that must be restructured according to the new schema, possibly through some kind of ETL process.
3. The application programs, that must interface with the new schema and comply with the new API. This generally involves rewriting some sections of the source code.

Each of these modifications follows its own rules, but we should not be surprised by the idea that the first one should strongly influence the others. This view currently is emerging under the name *co-transformation* [30]. Indeed, it has been proved that it is

possible to automatically derive data transformations (ETL) directives, as a SQL script for instance, from schema transformations [27]. Program transformation is much more complex. Automating this conversion has been studied in [26] and [9], and has been proved to be feasible.

One of the arguments of this paper is that one can study all transformations, including inter-model transformations, in the framework of a single model[3]. This raises the question of the nature of this generic model. Two approaches have been proposed, that distinguish themselves by the granularity of the model [24].

One approach, that can be called *minimalistic*, or *bottom-up*, relies on a very simple and abstract formalism, from which one can define more elaborated and richer models. Such a model generally represents the schema constructs specific to a definite model by abstract nodes and edges, together with assembly constraints. AutoMED [7] is a typical representative of this approach.

Another approach, symmetrically called *maximalistic* or *top-down*, is based on a large spectrum model, that includes, though in an abstract form, the main constructs of the set of operational models that are used in the engineering processes. The GER model follows this principle. It has been described in [21] and [24], and will be the basis of this paper.

3 Modeling Data Structures

3.1 Dealing with Multiple Models

Some database engineering processes transform schemas from one model to itself, involving one model only. Such is the case of relational normalisation, and of XML manipulation. These processes make use of *intra-model* transformations. Being dedicated to this model, their form generally is quite specific (e.g., respectively relational algebra and XSLT) and cannot be reused for other models.

Other processes, on the contrary, produce a result that is expressed into a model that is different from that the source schema. The most obvious example is the so-called *database logical design*, the goal of which is to transform an abstract conceptual schema into an operational (say, relational) logical schema as will be discussed in Section 6.2. In such cases one makes use of *inter-model* transformations. Many comprehensive processes, such as database design, reverse engineering and integration involve several abstraction levels and several technologies (and therefore models).

To master this complexity, several approaches rely on some kind of **pivot** model. The idea is quite simple, and has been adopted as an elegant way to solve the combinatorial explosion in situations in which mappings must be developed from any of M formalisms to any of N formalisms. Theoretically, one would need N x M distinct mappings. Thanks to the introduction of a *intermediate* or *pivot* formalism P, one needs to define M + N mappings only. Language translation and plateform-independent components are two of the most common examples.

[3] As illustrated in Fig. 1.

In the database engineering realm, dealing with a dozen models is not uncommon in large organizations. Developing, migrating, integrating, reverse engineering databases or publishing corporate data on the web all are processes that require inter-model schema transformation and, accordingly, data conversion. Considering N operational models, and admitting that the mappings among any pair of models are potentially useful in some processes, we need to define N^2 mappings, while the introduction of a pivot model allows us to reduce the number of mappings to $2 \times N + 1$. Fig. 2 identifies the mappings that will, sooner or later, be useful in an organization in which the data are stored in CODASYL and relational databases, that describes its information needs through Entity-relationship schemas, and that produces XML documents. Sixteen inter-model mappings are necessary, while the pivot model reduces the number of mappings to nine only. Moreover, all the mappings but one serve the mere function of formalism conversion ($\Sigma m>m'$, with $m \neq m'$), and therefore are fairly simple, while the power needed to express complex data structure transformation is the responsibility of one mapping only, namely $\Sigma p>p$. Introducing any new operational model M implies the development of two additional mappings $\Sigma m>p$ and $\Sigma p>m$.

An interesting consequence of approaches based on a pivot model is that inter-model transformations reduce to intra-model ones.

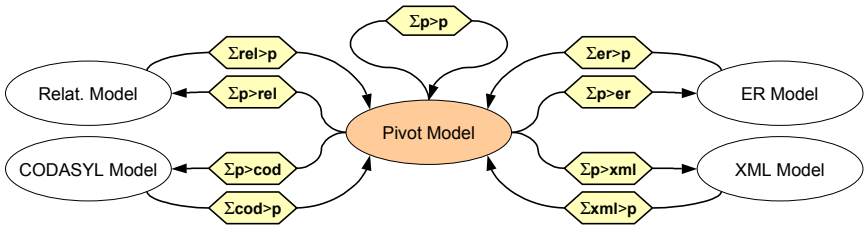


Fig. 2. Introducing a pivot model among N models reduces the number of inter/intra-model mappings

The example of *relational logical design*, that is, producing a relational schema from a conceptual schema, is illustrated in Fig. 3, which is just a subset of Fig. 2. It reads as follows:

1. the source conceptual schema is transformed into the pivot model ($\Sigma er>p$),
2. the resulting schema is transformed through a set of rules ($\Sigma p>p$) such as those that are largely described in the literature (see [4] for example[4]);
3. finally, the transformed schema obtained is expressed into the target relational model ($\Sigma p>rel$).

The next section describes in an informal way the main constructs of a pivot model on which we will base our discussion, namely the GER model.

[4] Not the most recent reference actually, but still one of the best.

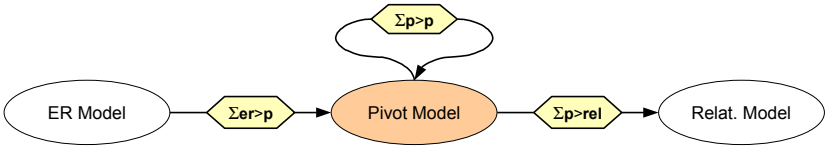


Fig. 3. Modeling *relational logical design* with a pivot model

Remark. The interpretation of Fig. 2, 3 and some of those that follow, needs to be precised a bit further. All schemas that can be expressed in model M are represented by the M-labelled ellipse. The mapping Σm>m' states that any schema expressed in the source model M is transformed through Σm>m' into a schema that complies with the target model M'.

3.2 The Generic Entity-relationship Model (GER)

The GER model, GER for short, is an extended Entity-relationship model that includes, among others, the concepts of schema, entity type, domain, attribute, relationship type, keys, as well as various constraints. In this model, a schema is a description of data structures. It is made up of specification constructs which can be, for convenience, classified into the usual three abstraction levels, namely conceptual, logical and physical. We will enumerate some of the main constructs that can appear at each level (Fig. 4).

- A *conceptual schema comprises* entity types (with/without attributes; with/without identifiers), super/subtype hierarchies (single/multiple; total and disjoint properties), relationship types (binary/N-ary; cyclic/acyclic; with/without attributes; with/without identifiers), roles of relationship type (with min-max cardinalities[5]; with/without explicit name; single/multi-entity-type), attributes (of entity or relationship types; multi/single-valued; atomic/compound; with cardinality[6]), identifiers (of entity type, relationship type, multivalued attribute; comprising attributes and/or roles), constraints (inclusion, exclusion, coexistence, at-least-one, etc.)
- A *logical schema comprises* record types, fields, arrays, single-/multi-valued foreign keys, redundant constructs, etc.
- A *physical schema comprises* files, record types, fields, access keys (a generic term for index, calc key, etc), physical data types, bag/list/array multivalued attributes, and other implementation details.

[5] The role cardinality constraint, denoted by i-j, specifies the range of the number of relationships in which an entity can appear in a definite role. Value N of j denotes infinity.

[6] Same as *role cardinality* applied to the number of attribute values *per* parent instance.

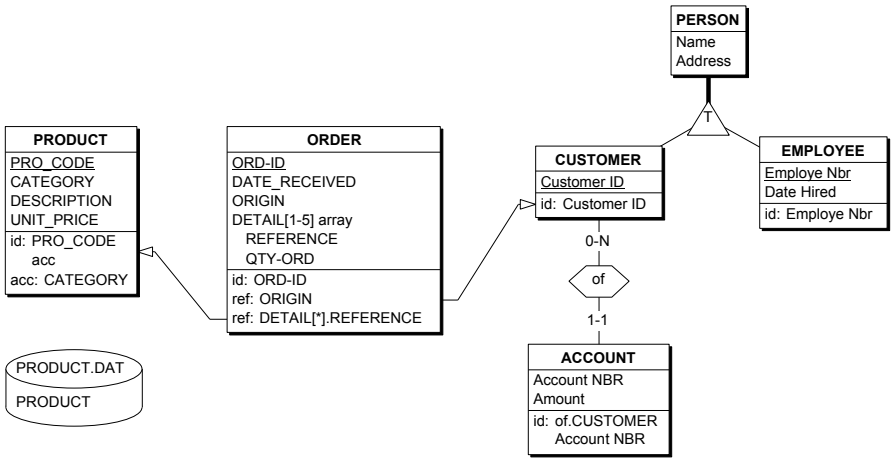


Fig. 4. A typical hybrid schema made up of conceptual constructs (e.g., entity types *PERSON*, *CUSTOMER*, *EMPLOYEE* and *ACCOUNT*, relationship type *of*, identifiers *Customer ID* of *CUSTOMER*), logical constructs (e.g., record type *ORDER*, with various kinds of fields including an array, foreign keys *ORIGIN* and *DETAIL.REFERENCE*) and physical objects (e.g., table *PRODUCT* with primary key *PRO_CODE* and indexes *PRO_CODE* and *CATEGORY*, table space *PRODUCT.DAT*). Note that the identifier of *ACCOUNT*, stating that the accounts of a customer have distinct account numbers (*Account NBR*), makes it a *dependent* or *weak entity type*.

3.3 Formal Semantics of the GER

In this paper, we develop transformational operators and discuss their properties. Many approaches rely on some intuitive rewriting rules expressed graphically. Though this is quite appropriate to allow readers to grasp the idea of the operators, a more formal treatment is necessary. In particular, we must base the definition and the evaluation of the properties of each operator on a rigorous basis, that is, a formal semantics of the GER. This is important for at least two reasons. First, formal semantics allows us to reason about transformations, and in particular to state its main properties such as the preservation of the information capacity of the source schemas. Secondly, implementing a set of transformations, for instance in a CASE tool, must rely on a completely defined semantics of both the model and the operators.

In Part 2, Sections 9 and 10, we will give the GER a precise semantics by stating mapping rules between the constructs of the GER and constructs of a variant of the N1NF relational formalism, called Extended Relational Model (ERM). Each GER construct will be given an ERM interpretation, and, conversely, each construct of the ERM will be given a GER interpretation. Basically, these mappings are the inter-model transformations $\Sigma_{ger>erm}$ and $\Sigma_{erm>ger}$ depicted in Fig. 5. The ERM is described in Section 9 while mapping $\Sigma_{ger>erm}$ and its inverse are presented in Section 10. The reader will find a complete formalization of the GER in [16].

[44] follows another approach. The author associates with HERM, a variant of the ER model, a specific notation with a precise ad hoc semantics, that includes an algebra and a calculus.

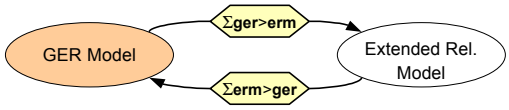


Fig. 5. Expressing the semantics of the GER model by a bidirectional mapping with the Extended Relational Model (ERM)

Note. The interpretation of the inverse mapping Σerm>ger is a bit more complex than suggested. Indeed, Σger>erm is **not surjective**, so that some ERM schemas have no GER counterpart. To be quite precise, we should define the subset ERM* of ERM that makes Σger>erm surjective. However, we will ignore this for simplicity sake. This is no problem since ERM* is closed under the set of ERM transformations Σerm>erm that we will use[7]. Proving this is fairly easy but would lead us beyond the scope of this paper. In the rest of this paper, we will admit that the composition Σerm>ger ∘ Σger>erm is the identity mapping without loss of generality.

3.4 Specifying Operational Models in the GER

Popular operational formalisms, that is, those which are in practical use among developers, can be expressed as *specializations* of the GER. In general, deriving model M from model M0 (here the GER) consists in,

1. selecting the constructs of M0 that are pertinent in M;
2. specifying the structural constraints on this subset so that only schemas valid in M can be built;
3. renaming these constructs in order to make them compliant with the taxonomy of M; this step will be ignored in this paper.

This process materializes the mapping ΣM>M0. We will briefly discuss this mapping for two models, namely Entity-relationship model and the SQL2 relational model (Fig. 6). Similar mapping can be (and have been) developed for CODASYL and IMS models, for standard files structures, and for XML DTDs and Schemas.

Fig. 6. Two mappings described in Sections 3.5 and 3.6

[7] Σerm>erm is the set of ERM-to-ERM transformations. Applying operators from the subset of Σerm>erm that underlies Σger>ger (as discussed in Section 12) to any ERM* schema produces an ERM* schema.

3.5 GER Expression of the Entity-relationship Model

Let us first observe that there is no such thing as a *standard ER model*. At least a dozen formalisms have been proposed, some of them being widely used in popular text books and in CASE tools. However, despite divergent details, they all share essential constructs such as entity types, relationship types with roles, some kind of role cardinality/multiplicity constraints, attributes and unique keys. Due to the nature of the GER, restricting it to a definite Entity-relationship model is fairly straighforward, so that we do not propose to develop the Σer>ger mapping.

The increasing popularity of the UML class model[8] (*aka* class diagrams) incites some authors and practitioners to use them to specify database conceptual and logical schemas. This was not the primary objective of the UML formalism, so that it exhibits severe flaws and weaknesses in database modelling. However, mapping Σuml>ger can be developed in the same way as for other models.

3.6 GER Expression of the Standard Relational Model (SQL2)

A relational schema mainly includes tables, domains, columns, primary keys, unique constraints, not null constraints and foreign keys. The relational model can therefore be defined as in Fig. 7.

relational constructs	GER constructs	assembly rules
database schema	schema	
table	entity type	an entity type includes at least one attribute
domain	simple domain	
nullable column	single-valued and atomic attribute with cardinality [0-1]	
not null column	single-valued and atomic attribute with cardinality [1-1]	
primary key	primary identifier	a primary identifier comprises attributes with cardinality [1-1]
unique constraint	secondary identifier	
foreign key	reference group	the composition of the reference group must be the same as that of the target identifier
SQL names	GER names	the GER names must follow the SQL syntax

Fig. 7 - Defining the standard relational (SQL2) model as a subset of the GER model (mapping Σrel>ger).

[8] The term UML model has a specific interpretation in UML, where it denotes what we call a schema in this paper.

A GER schema made up of constructs from the second column only, and that satisfies the assembly rules stated in the third column, can be called a *relational GER schema*. As a consequence, a relational schema cannot comprise *is-a* relations, relationship types, multivalued attributes nor compound attributes.

4 Schema Transformation

Let us denote by M the unique model in which the source and target schemas are expressed, by S the schema on which the transformation is to be applied and by S' the schema resulting from this application. Let us also consider sch(M), a function that returns the set of all the valid schemas that can be expressed in model M, and inst(S), a function that returns the set of all the instances that comply with schema S.

4.1 Specification of a Transformation

A **transformation** Σ consists of two mappings **T** and **t** (Fig. 8):

1. **T** is the *structural mapping* from sch(M) onto itself, 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 the *weakest precondition* **P** that any construct C must satisfy in order to be transformed by T, and the *strongest postcondition* **Q** that T(C) satisfies. **T** specifies the rewriting rule of Σ.
2. **t** is the *instance mapping* from inst(S) onto inst(S'), 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** can be specified through any algebraic, logical or procedural expression.

According to the context, Σ will be noted either **<T,t>** or **<P,Q,t>**.

The nature of the most suited formalism in which P, Q and t could be expressed[9] will not be discussed here. In the following, we will use abstract schema fragments following the graphical convention of the underlying model.

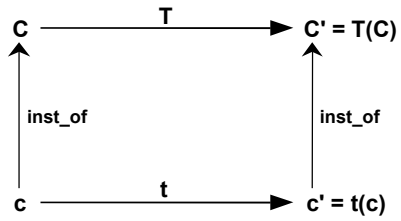


Fig. 8. The two mappings of schema transformation S ≡ <T,t>. The inst_of arrow from x to X indicates that x is an instance of X.

[9] Description logic [2] could be a good candidate for P and Q.

4.2 Generic, Parametric and Instantiated Transformations

Let us consider relation R, the attributes of which are partitioned into non empty subsets I, J and K. Considering Σ, a (lossy) variant of relational decomposition transformation, predicates P and Q as well as instance transformation t can be written as follows:

T	**P**	R(U); {I,J,K} partition of U;
	Q	R1(IJ); R2(IK);
t		*let r be the current instance of R; let r1, r2 be instances of R1, R2*; r1 = r[IJ]; r2 = r[IK];

Σ is *generic*, since it gives an abstract pattern that must be applied to an actual relation before being carried out. Let us apply Σ to relation CUST(C#, CNAME, CADD, CACC) of a supposedly current schema. We observe that there are several ways to instantiate Σ according to the values we assign to variables I, J and K, leading to as many *instantiated* transformations. For this reason, we call Σ a *parametric* transformation. For instance, with assignments I := {CNAME} and J := {C#, CADD}, we get the following fully *instantiated* transformation.

T	**P**	CUST(C#, CNAME, CADD, CACC); I = {CNAME}; J = {C#, CADD} ;
	Q	C1(CNAME, C#, CADD); C2(CNAME, CACC);
t		*let c be the current instance of CUST; let c1, c2 be instances of C1, C2;* c1 = c[CNAME, C#, CADD]; c2 = c[CNAME, CACC];

A generic transformation can be partially instantiated if some, but not all, variables of P have been instantiated.

Each transformation Σ is associated with an *inverse transformation* Σ' which can undo the result of the former under certain conditions that will be detailed in the next section.

4.3 Semantics Preservation Properties of Transformations

One of the most important properties of a transformation is the extent to which the target schema can replace the source schema without loosing information. This property is called *semantics preservation* or *reversibility*.

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 equivalent 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.

Similarly, in the pure software engineering domain, [3] introduces the concept of *correctness-preserving* transformation aimed at compilable and efficient program production.

We must consider two different classes of transformations, namely reversible and symmetrically reversible.

1. A transformation Σ1 = <T1,t1> = <P1,Q1,t1> is *reversible*, *iff* there exists a transformation Σ2 = <T2,t2> = <P2,Q2,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 the converse is not true. For instance, an arbitrary instance c' of T(C) may not satisfy the property c'=t1(t2(c')).
2. If Σ2 is reversible as well, then Σ1 and Σ2 are called *symmetrically reversible*. In this case, Σ2 = <Q1,P1,t2>. Σ1 and Σ2 are called *SR-transformations* for short.

Example

The so-called *decomposition theorem* of the 1NF relational theory [13] is an example of reversible transformation that can be described as follows[10].

T1	**P1**	R(U); {I,J,K} partition of U; I →→ J|K;
	Q1	R1(IJ); R2(IK);
t1		*let r be the current instance of R; let r1, r2 be instances of R1, R2*; r1 = r[IJ]; r2 = r[IK];

However, there is no reason for any arbitrary couple of instances r1 of R1 and r2 of R2 to enjoy the inverse property r = (r1*r2)[IJ]. We must refine this transformation in order to make it symmetrically reversible. This transformation and its inverse are summarized here below.

T1	**P1**	R(U); {I,J,K} partition of U; I →→ J|K;
	Q1	R1(IJ); R2(IK); R1[I] = R2[I];
t1		*let r be the current instance of R; let r1, r2 be instances of R1, R2*; r1 = r[IJ]; r2 = r[IK];
t2		let r1, r2 be current instances of R1, R2; let r be an instance of R; r = r1*r2[IJK];

4.4 Generating and Studying GER Transformations

The complexity of high-level models, and that of the GER in particular, makes the study of their transformations particularly complex. To begin with, experience shows that several dozens of operators can be useful, if not necessary, to describe the most important engineering processes. Then, identifying and proving the reversibility degree of each of them can be a huge and complex task, notably since there is no agreed upon algebra or calculus to express Entity-relationship queries.

[10] →→ denotes a multivalued dependency, [ ] the projection operator and * the join operator.

The key lies in the ERM formalism that expresses the semantics of the GER. Indeed, the relational model, of which the ERM inherits, includes a strong and simple body of properties and inference rules that can be used to built a relational transformational theory. We can reasonably expect the set of transformations defined for the ERM to be far smaller and simpler than that of the GER.

If this idea proves to be correct, then we will be provided with a nice way to generate, explain, and reason on, GER transformations. Fig. 9 illustrates this approach.

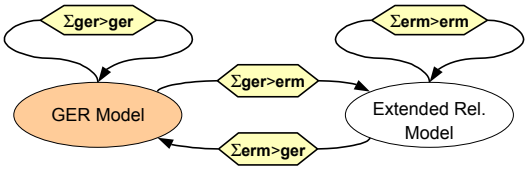


Fig. 9. Generating and specifying GER transformations through their expression in the Extended Relational Model

According to this view, each GER transformation can be modelled as the compound mapping:

$$\Sigma ger{>}ger = \Sigma erm{>}ger \circ \Sigma erm{>}erm \circ \Sigma ger{>}erm$$

Since Σerm>ger and Σger>erm are symmetrically reversible, a transformation in Σger>ger is semantics-preserving *iff* there exists a (possibly compound) transformation in Σerm>erm that is symmetrically reversible. Section 11 describes the main transformations of Σerm>erm. Then, Section 12 interprets three popular GER transformations as compound ERM transformations.

5 Typology of Practical Transformations

This section describes several families of GER transformations with which complex engineering processes will be built.

5.1 Mutation Transformations

A mutation is an SR-transformation that changes the nature of an object. Considering the three main natures of object, namely *entity type*, *relationship type* and *attribute*, six families of mutation transformations can be defined. Fig. 10 shows the structural mapping (T) of some representative operators (couples of operators Σ1 to Σ3) applied to typical schema fragments. The transformations Σ4 are not primitive since they can be defined by combining other mutations. However, they have been added due to their usefulness.

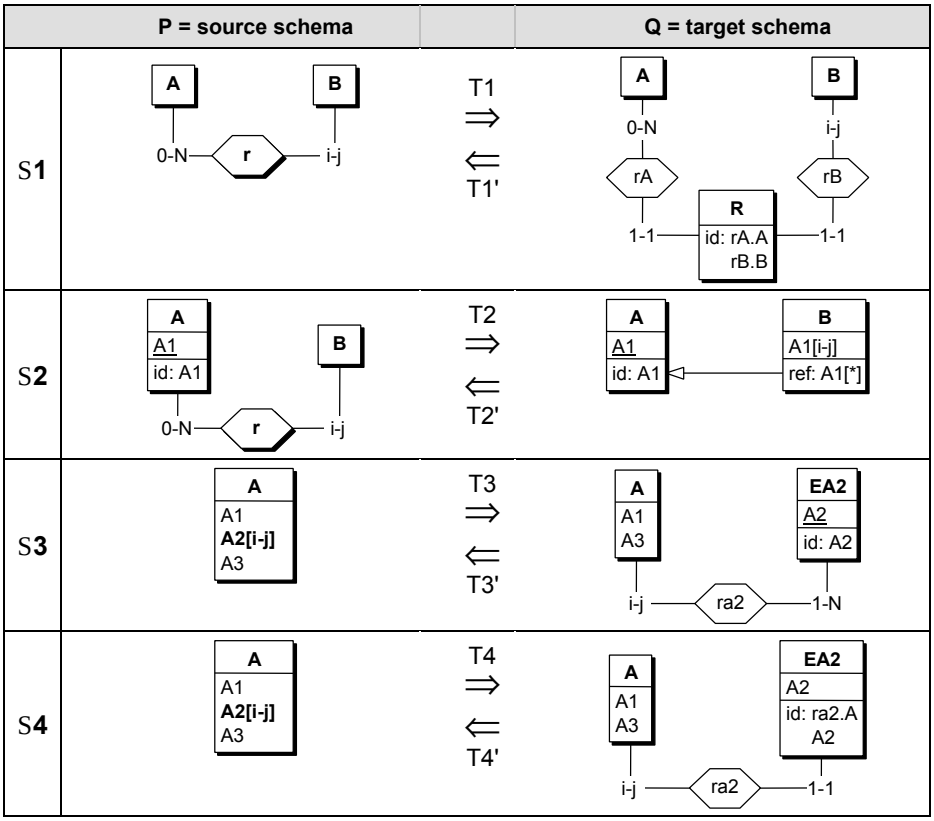


Fig. 10 - Six representative mutation transformations Σ1 to Σ3. Transformations Σ1 generalized to N-ary rel-types as will be shown in Fig. 26. Though not primitive, compound transformations Σ4 are shown as well. Cardinality constraints [i-j] are arbitrary values.

5.2 Other Elementary Transformations

The mutation transformations can solve many database engineering problems, but other operators are needed to model special situations.

Expressing supertype/subtype hierarchies in DMS that do not support them explicitly is a recurrent problem. The technique of Fig. 11 is one of the most commonly used [4] [23]. It consists in representing each source entity type by an independent entity type, then to link each subtype to its supertype through a one-to-one relationship type. The latter can, if needed, be further transformed into foreign keys by application of Σ2-direct.

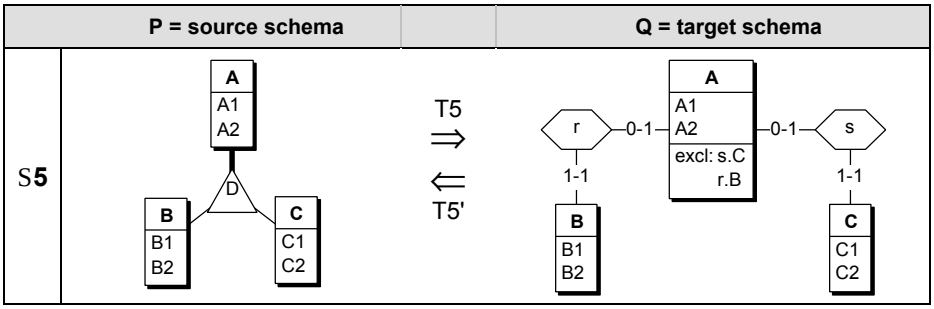


Fig. 11 - Transforming an is-a hierarchy into one-to-one relationship types and conversely. The exclusion constraint (excl:s.C,r.B) states that an A entity cannot be simultaneously linked to a B entity and a C entity. It derives from the disjoint property (D) of the subtypes.

Transformations Σ3 and Σ4 showed how to process standard multivalued attributes. When the collection of values is no longer a set but a *bag*, a *list* or an *array*, operators to transform them into standard multi-valued attributes are most useful. Transformations Σ6 in Fig. 12 are dedicated to arrays. Similar operators have been defined for the other types of containers.

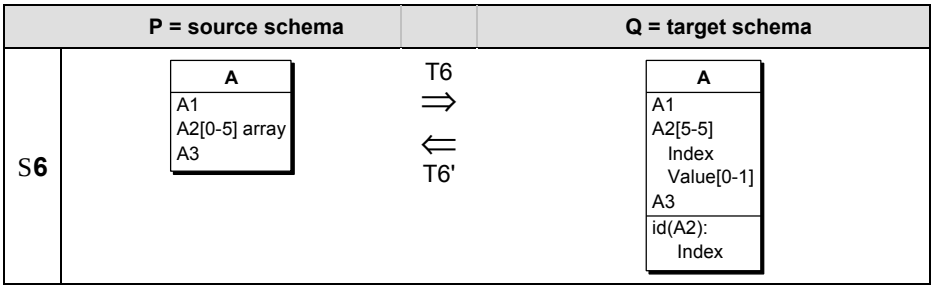


Fig. 12 - Converting an array A2 into a set-multivalued attribute and conversely. The values are distinct wrt component Index (id(A2):Index). The latter indicates the position of the cell that contains the value (Value). The domain of Index is the range [1..5].

Attributes defined on the same domain and the name of which suggests a spatial or temporal dimension (e.g., departments, countries, years or pure numbers) are called *homogeneous serial attributes*. In many situations, they can be interpreted as the representation of an indexed multivalued attributes (Fig. 13). The identification of these attributes must be confirmed by the analyst.

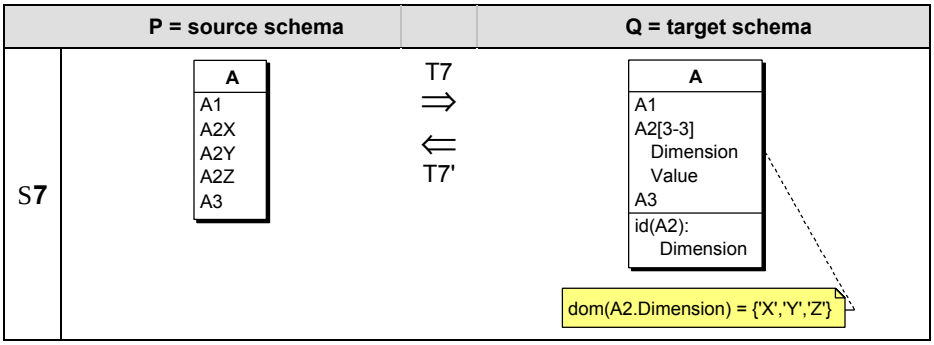


Fig. 13 - Transforming homogeneous serial attributes {A2X, A2Y, A2Z} into a multivalued compound attribute A2 and conversely. The values (Value) are indexed with the distinctive suffix of the source attribute names, interpreted as a dimension (sub-attribute Dimension).

5.3 Compound Transformations

A compound transformation is made up of a chain of more elementary operators in which each transformation applies on the result of the previous one. The transformation Σ8 in Fig. 14, illustrated by a concrete example, transforms a complex relationship type R into a sort of *bridge* entity type comprising as many foreign keys as there are roles in R. It is defined by the composition of Σ1-direct (generalized to N-ary rel-types) and Σ2-direct. This operator is of frequent use in relational database design.

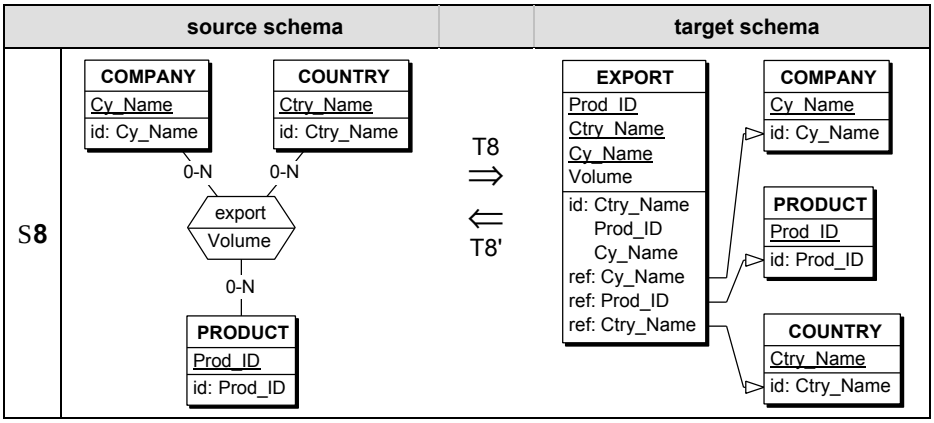


Fig. 14 - Transformation of a complex relationship type into relational structures.

The transformation Σ9 is more complex (Fig. 15). It is composed of a chain of four elementary operators. The first one transforms the serial attributes Expense-2000, ..., Expense-2004 into multivalued attribute Expense comprising subattributes Year (the dimension) and Amount (transformation Σ7-direct). The second one extracts this

attribute into entity type EXPENSE, with attributes Year and Amount (transformation Σ4-direct). Then, the same operator is applied to attribute Year, yielding entity type YEAR, with attribute Year. Finally, entity type EXPENSE is transformed into relationship type expense (Σ1-inverse).

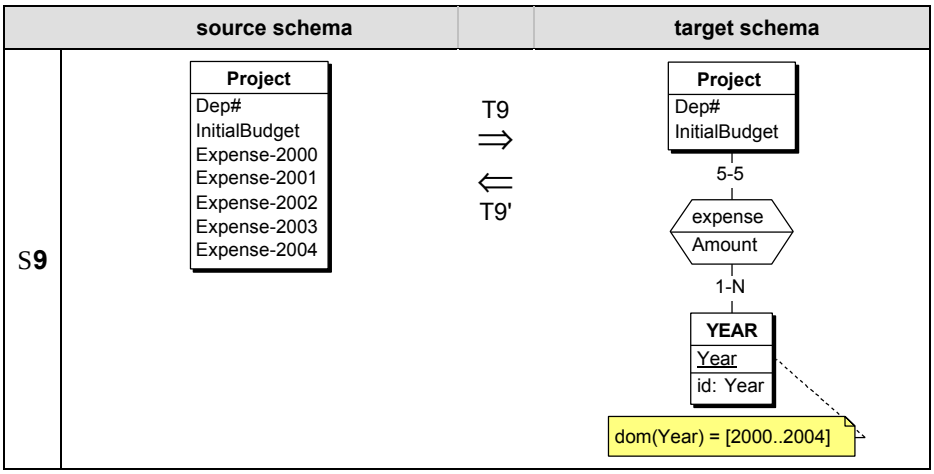


Fig. 15 - Extracting a temporal dimension from homogeneous serial attributes.

5.4 Predicate-Driven Transformations

A predicate-driven transformation Σp applies an operator Σ to all the schema objects that meet a definite predicate p.

predicate-driven transformation	interpretation
RT_into_ET(ROLE_per_RT(3 N))	transform each rel-type R into an entity type (RT_into_ET), if the number of roles of R (ROLE_per_RT) is in the range [3 N]; in short, *convert all N-ary rel-types into entity types.*
RT_into_REF(ROLE_per_RT(2 2) and ONE_ROLE_per_RT(1 2))	transform each rel-type R into reference attributes (RT_into_REF), if the number of roles of R is 2 and if R has from 1 to 2 "one" role(s), i.e., R has at least one role with max cardinality 1; in short, *convert all one-to-many rel- types into foreign keys.*
INSTANTIATE(MAX_CARD_of_ATT(2 4))	transform each attribute A into a sequence of single-value instances, if the max cardinality of A is between 2 and 4; in short, *convert multivalued attributes with no more than 4 values into serial attributes.*

Fig. 16 - Three examples of predicate-driven transformations. *Rel-type* is a short-hand for *Relationship type*.

It will be specified by $\Sigma(p)$. p is a *structural* predicate that states the properties through which a class of patterns can be identified. In general, the inverse of Σp cannot be derived from the expression of Σ and p. Indeed, there is no means to derive the predicate p' that identifies the constructs resulting from the application of Σp, and only them.

We give in Fig. 16 some useful transformations that are expressed in the specific language of the DB-MAIN tool (Section 8), which follows the $\Sigma(p)$ notation. Most predicates are parametric. For instance, the predicate ROLE_per_RT(n m), where n and m are integers such that $n \leq m$, states that the number of roles of the relationship type falls in the range [n..m]. The symbol "N" stands for infinity.

5.5 Model-Driven Transformations.

A model-driven transformation is a goal-oriented compound transformation made up of predicate-driven operators. It is designed to transform any schema expressed in model M into an equivalent schema in model M'.

As illustrated in the discussion of the relational model expressed as a specialization of the GER (Fig. 7), identifying the components of a model also leads to identifying the constructs of the GER that do not belong to it. Except when M is a subset of M', an arbitrary schema $S \in sch(M)$ may include constructs that violate M'. Each class of constructs that can appear in a schema can be specified by a structural predicate. Let P_M denote the set of predicates that defines model M and $P_{M'}$ that of model M'. In the same way, each potentially invalid construct can also be specified by a structural predicate. Let $P_{M/M'}$ denote the set of predicates that identify the constructs of M that are not valid in M'. In the DB-MAIN language used in Fig. 16, ROLE_per_RT(3 N) is a predicate that identifies N-ary relationship types that are known to be invalid in DBTG CODASYL schemas, while MAX_CARD_of_ATT(2 N) defines the family of multivalued attributes, that is invalid in the SQL2 database model. Finally, we observe that a set such as P_M can be rewritten as a single predicate formed by *anding* its components.

Let us now consider predicate $p \in P_{M/M'}$, and let us choose a transformation $\Sigma = \langle P,Q,t \rangle$ such that,

$$(p \Rightarrow P) \wedge (P_{M'} \Rightarrow Q)$$

Clearly, the predicate-driven transformation $\Sigma(p)$ solves the problem of the invalid constructs defined by p. Proceeding in the same way for each component of $P_{M/M'}$ provides us with a series of operators that can transform any schema in model M into schemas in model M'. We call such a series a *transformation plan*, which is the practical form of any model-driven transformation. In real situations, a plan can be more complex than a mere sequence of operations, and may comprise loops to process recursive constructs for instance. Transformation plans implement what some authors call *strategies*, that is, deterministic or heuristic reasoning on how to apply transformations to reach a definite goal. [1] propose strategies to convert VDM data types in relational structures while [40] applies semi-procedural strategies to high-level engineering processes.

In addition, transformations such as those specified above may themselves be compound, so that the set of required transformations can be quite large. In such cases, it can be better to choose a transformation that produces constructs that are not fully compliant with M', but that can be followed by other operators which complete the job. For instance, transforming a multivalued attribute into relational structures can be obtained by an ad hoc elementary transformation. However, it can be thought more convenient to first transform the attribute into an entity type + a one-to-many relationship type (Σ4-direct), which can then be transformed into a foreign key (Σ2-direct). This approach produces transformation plans which are more detailed and therefore less readable, but that rely on a smaller and more stable set of elementary operators.

The transformation toolset of DB-MAIN includes about thirty operators that have proved sufficient to process schemas in a dozen operational models. If the transformations used to build the plan have the SR-property, then the model-driven transformation that the plan implements is symmetrically reversible. When applied to any source schema, it produces a target schema semantically equivalent to the former.

6 Modeling Standard Database Engineering Processes as Transformations

Complete database engineering processes, such as database development, database reverse engineering, data warehouse design or database migration comprise several steps, most of which can be viewed as chains of transformations, or, more specifically, transformation plans. This section illustrates the issue by modeling one of the major processes, namely *database logical design*, through the transformational paradigm.

6.1 Database Design

The process of designing and implementing a database that is to meet definite users requirements has been described extensively in the literature [4] and has been available for several decades in CASE tools. It comprises four main sub-processes, namely (Fig. 17):

1. *Conceptual design*, the goal of which is to translate users requirements into a conceptual schema, which is a technology-independent abstract specification[11].
2. *Logical design*, which produces a logical schema that losslessly translates the constructs of the conceptual schema according to a specific technology family[12].
3. *Physical design*, which augments the logical schema with performance-oriented constructs and parameters, such as indexes, buffer management policies or lock management parameters.

[11] or *Platform-Independent Model* (PIM according to the MDA/MDE vocabulary.

[12] The logical and physical schemas can be called *Platform-Specific Model* (PSM in the MDA/MDE vocabulary).

4. *Coding*, that translates the physical schema (and some other artefacts) into the DDL code of the DBMS.

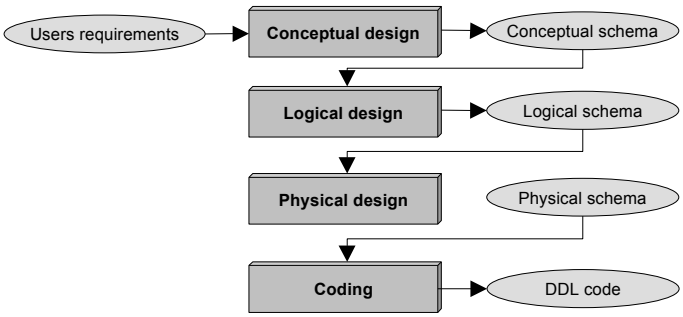


Fig. 17 - The main processes of database design

Calling the whole process DB-Design, and the four sub-processes respectively ConcD, LogD, PhysD and Coding, we can describe them with the transformational notation:

DDL code = DB-design(Users requirements)

DB-design = Coding ∘ PhysD ∘ LogD ∘ ConcD

These processes are model-driven transformations that can be described by transformation plans. The level of formality of these processes depends on the methodology, on the existence of CASE support and of non functional requirements such as performance and robustness, that generally require human expertise. For instance, conceptual design is a highly informal process based on human interpretation of complex information sources, while logical design can be an automated process completely described by a transformation plan. Anyway, these processes can be decomposed into sub-processes that, in turn, can be modelled by transformations and described by transformation plans, and so forth, until the latter reduce to elementary operators such as those described in Sections 5.1 and 5.2. Below, we examine the Logical design process in further detail.

6.2 Database Logical Design

We consider the most popular conceptual source model, namely the Entity-relationship model, and the most popular logical target model, the SQL2 relational model, to which Oracle, SQL Server, DB2, PostgreSQL, Firebird and many others are compliant. The GER expression of the SQL2 model has been developed in Fig. 7. By complementing this table, we identify the Entity-relationship constructs that **do not belong** to the SQL2 model, the four most important of which being transformed as follows.

Transforming *is-a* relations
Transformation Σ5-direct eliminates this structure without semantics loss by introducing one-to-one (functional) rel-types. The latter can then be processed by the mutation transformation Σ2-direct that generates foreign keys.

Transforming *relationship types*
Two cases must be considered. The easy case is that of *functional rel-types*, that can be replaced by foreign keys through transformation Σ2-direct.

The complex patterns comprise *non-functional rel-types*, that is, those which are *many-to-many*, or *N-ary*, or which *have attributes*. They are first transformed into entity types with operator Σ1-direct. Then, the resulting functional rel-types are transformed into foreign keys (Σ2-direct). Note that the whole process is a compound transformation that has been described as Σ8-direct.

Transforming *multivalued attributes*
A multivalued attribute that directly depends on its parent entity type (level 1) is transformed into an entity type, through the compound mutation operator Σ4-direct. If the attribute is compound, it is suggested to incorporate its components in the new entity type, and not the attribute itself. This generates a one-to-many rel-type, that is further transformed into a foreign key.

Transforming single-valued *compound attributes*
The simplest way to transform a level 1 compound attribute is to replace it with its components, a technique called *disaggregation* (transformation Σ10-direct, not illustrated). Another technique consists in processing the attribute as if it was multivalued as described here above (Σ4-direct). In this case, it is transformed into an entity type and a functional rel-type, itself transformed into a foreign key.

Note on the transformation of a rel-type into a foreign key
This transformation requires the other entity type to have an identifier made up of attributes. Otherwise, we have to give it a *technical identifier* (transformation Σ11-direct, not illustrated).

Grouping similar transformations and reorganizing the operations logically provides us with a simple but fairly powerful transformation plan that transforms most conceptual Entity-relationship schemas into pure relational schemas (Fig. 18). Since we have used SR-transformations only, the whole process is semantics-preserving[13]. Actual plans are more complex, but follow the same approach. Let us mention some extensions: eliminating optional identifiers, other techniques to implement is-a rela-

[13] This assertion is not quite correct if we only use the transformations presented in this paper. In particular, some constraints can be lost, or incompletely translated. Such is the case for cardinality constraints [i-j] where 1 < j < N. A more comprehensive plan, making use of more precise transformations, can preserve these constraints until the coding phase, e.g., in the form of SQL triggers.

tions (e.g., by descending or ascending inheritance), instantiating multivalued attributes, concatenating multivalued attributes, concatenating compound attributes, etc.

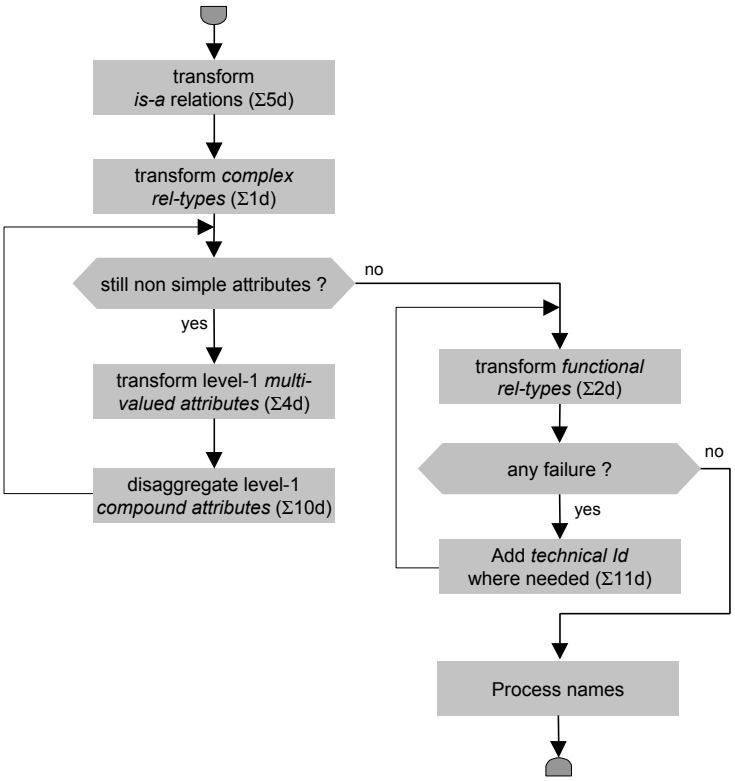


Fig. 18 - A simple transformation plan for logical relational database design

6.3 Case Study

The conceptual schema of Fig. 19 includes, in a small footprint, several interesting constructs, such as complex rel-types, a cyclic rel-type, is-a relations, multivalued attributes, compound attributes, an entity type without identifier, an optional identifier, a mandatory *many* role (written.AUTHOR [1-N]) and a hybrid identifier.

The application of the transformation plan of Fig. 18, extended to the elimination of optional identifiers[14], produces the relational schema of Fig. 20.

[14] Several DBMS do not manage correctly candidate keys comprising a nullable column.

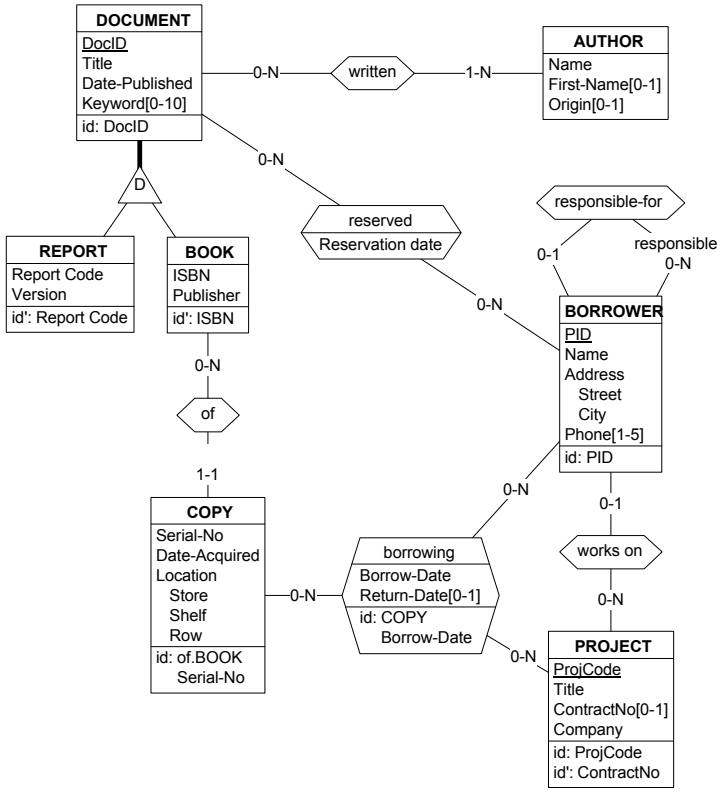


Fig. 19 - A representative conceptual schema

7 Modeling Database Reverse Engineering Process as Transformations

Many database engineering processes, such as maintenance, evolution, migration, integration or federation require the availability of a complete and up to date documentation, that is, for a database, its logical and conceptual schemas. Needless to say that these essential documents most often are missing, specially for legacy databases that can be more than 20 years old.

Database reverse engineering is the process through which one attempts to recover or to rebuild these schemas when they are missing, obsolete or incomplete. We will show that several important aspects of this process can be modelled by transformtions. Intensive research in the last decade have shown that reverse engineering generally is much more complex than initially thought.

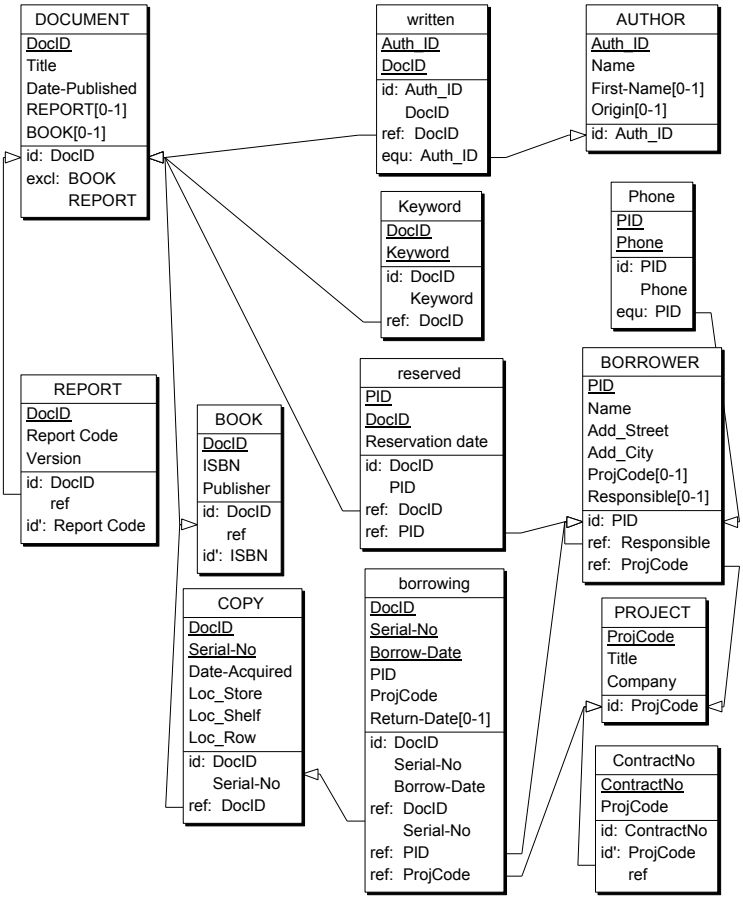


Fig. 20 - The relational schema obtained by the application of the transformation plan of Fig. 18 on the conceptual schema of Fig. 19

We can put forward three major sources of difficulties, namely (1) the absence of systematic design (empirical coding still is the most popular way to design a database), (2) the weaknesses of the legacy (and, paradoxically modern as well) DBMS, that force the developer to resort to various tricks to code the data structures and the integrity constraints and (3) only the DDL code provides a reliable description of the database physical constructs.

7.1 Database Reverse Engineering

In complex projects, for instance when the database includes several hundreds or thousands of tables[15], the core of the process will be organized as described in Fig. 21. It comprises four main sub-processes, namely:

[15] An SAP database can comprise 30,000 tables and more than 200,000 columns.

1. *Parsing*, that rebuilds the raw physical schema by merely parsing the DDL code ($code_{ddl}$). Only the constructs that have been *explicitly declared* in the code can be recovered.
2. *Refinement*, which enriches the raw physical schema with the *undeclared constructs* and constraints that have been elicited through the analysis of program code ($code_{prg}$), as well as other sources that we will ignore here. Sometimes more than 50% of the specifications can be retrieved in this way.
3. *Cleaning*, which removes the technical constructs, such as the indexes, and which produces the logical schema.
4. *Conceptualization*, which derives a plausible conceptual schema from the logical schema.

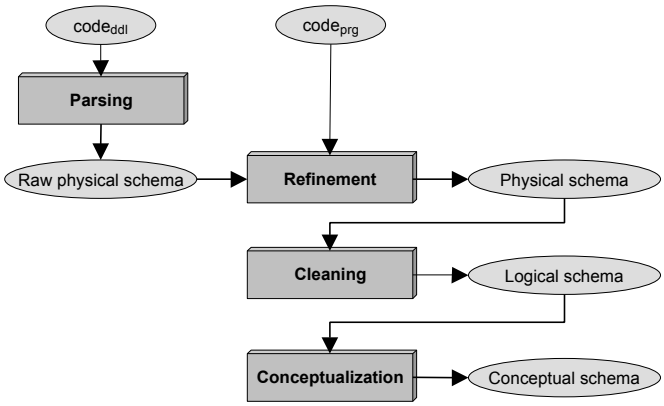


Fig. 21 - The four main processes of database reverse engineeering

Calling the whole process DB-REng, and the four sub-processes Parse, Refine, Clean and Concept respectively, we can write:

$$\text{Conceptual schema} = \text{DB-REng}(code_{ddl}, code_{prg})$$

$$\text{DB-design} = \text{Concept} \circ \text{Clean} \circ \text{Refine} \circ \text{Parse}$$

An interesting, and not really surprising, aspect of database reverse engineering is that all the processes we have mentioned appear to be the **reverse** of database design processes. Indeed, we have the following relations:

$$\text{Refine} \circ \text{Parse} = \text{Coding}^{-1}$$

$$\text{Clean} = \text{PhysD}^{-1}$$

$$\text{Concept} = \text{LogD}^{-1}$$

This observation has a deep influence on the specifications and the strategies of the reverse processes. For instance, since the Conceptualization process is the inverse of Logical design, it should be possible to derive a transformation plan for the former just by reversing the plan of Logical design. Though this approach has proved suc-

cessful, the problem is a bit more complex due to the undisciplined way legacy databases were designed. When the logical schema was built, it had to meet not only functional requirements (that is, to express all the semantics of the conceptual schema), but also non-functional requirements such as time-space optimization, security or privacy. The satisfaction of the latter requirements can deeply affect the readability of the logical schema to such an extent that it has become quite difficult to understand.

In the next section, we will very shortly describe the Conceptualization process as a transformation process, and elaborate a representative transformation plan.

7.2 Logical Schema Conceptualization

Reversing a transformation plan is a new concept that would deserve some further discussion [22]. Due to space limit, we will give a simplified definition that is valid for linear plans only, that is, plans which do not include if-then-else or loop constructs:

> *Considering a transformation S implemented by transformation plan T, T' is an inverse of T if it implements the inverse of S.*
>
> If T is a linear transformation plan, T' can be built as follows: *each operator of T is replaced with its inverse, then the resulting sequence is reversed.*

Deriving a linear plan from the plan proposed for Logical design in Fig. 18 is not too difficult, provided we target simpler schemas, that meet such realistic restrictions as the following: *a multivalued attribute can be compound, but no compound attributes can have components that are themselves compound or multivalued*. Fig. 22 depicts the linearized plan for *Logical design* (left), and a tentative transformation plan for *Logical schema Conceptualization* obtained by inverting the former (right).

The resulting plan introduces new processes and terms that deserve some explanation. Removing a technical Id is valid provided it does not represent any application domain concept. A series of heterogeneous serial attributes is a pattern in which a sequence of attributes, generally of different types, have names that present strong similarities, and that suggest that these attributes form an implicit aggregate (Example: Address-City, Address-Street, Address-Number). An attribute entity type AE is an entity type the goal of which obviously is just to add an elementary information to another entity type. It comprises one or a few attributes that are all part of the identifier of AE, and is linked to another entity type only, through a mandatory role. A relationship entity type is an entity type the role of which obviously is just to link two or more entity types. Transforming one-to-one rel-types into is-a relations must be carried out with caution, since it must be semantically pertinent. A one-to-one rel-type between MANAGER and CAR does not mean that CAR is a subtype of MANAGER!

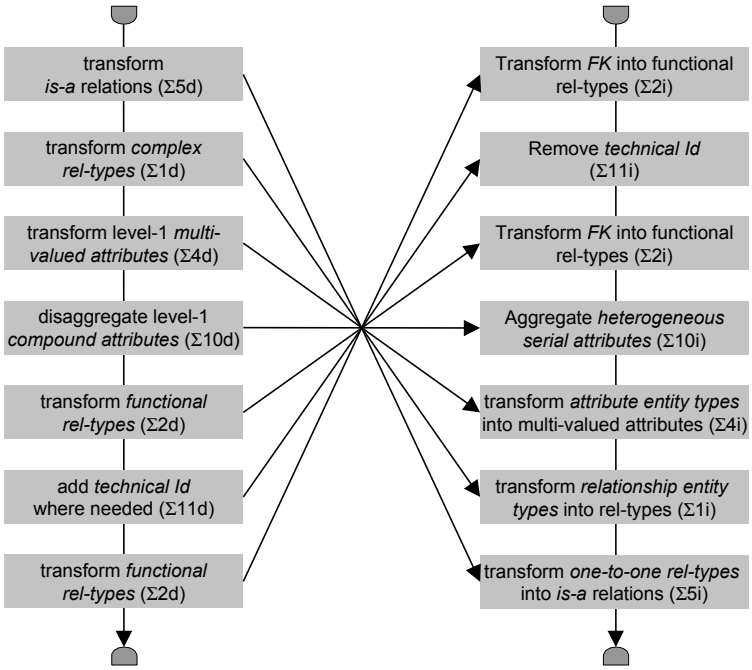


Fig. 22 - Building a linear transformation plan for the Conceptualization process

Finally, let us observe that the second step of the resulting transformation plan (right) is useless and can be discarded, though it does no harm[16].

7.3 Case Study

The application of this transformation plan to the logical relational schema of Fig. 20 is left as an exercice to the reader, preferably with the help of the Transformation assistant of the DB-MAIN CASE tool. Some observations:

1. identifying serial attributes forming attributes Location and Address is a manual process,
2. deleting the technical id of AUTHOR is a manual process,
3. the conceptual names of most one-to-many rel-types cannot be recovered (default names are suggested but they generally are not suitable), and must be assigned manually.

[16] A desirable property of these plans is their *idempotence*. It is not guaranteed in general.

8 Transformations in CASE Tools

Following the discussion of this paper, it is not surprising that the transformational paradigm is particularly suited to build CASE tools. All CASE tools rely, often implicitly, on some kind of schema transformations. Due to the popularity of the MDE approaches, we can expect future CASE tools to include programmable transformation toolsets. In the past, some examples of transformation-based tools have been described, e.g., in [42]. We can also mention Silverrun, a CASE tool that explicitly makes use of transformations.

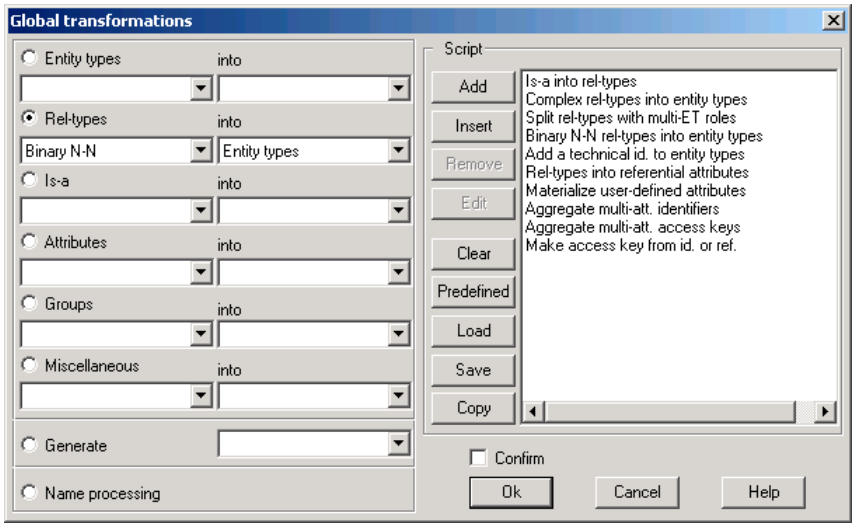


Fig. 23 - The elementary transformation assistant of DB-MAIN

We will describe briefly the transformation facilities of DB-MAIN[17], a CASE tool dedicated to the support of the main database engineering processes, including non standard ones, such as database reverse engineering, interoperability, active and temporal database design, wrapper generation and XML engineering. DB-MAIN is based on the GER model and offers a toolset of about 30 elementary transformations.

DB-MAIN includes a collection of programmable assistants that are intended to help the analysts in complex and tedious tasks. Two of them are of particular interest, namely the *Transformation assistant* and the *Advanced transformation assistant*. Both allow the analyst to apply predicate-driven transformations on the current schema and to build transformation plans through a scripting facility.

Fig. 23 shows a typical screen of the first assistant. Its left part proposes a list of labelled patterns (a *user-friendly* interface to built-in structural *predicates*), accompanied by a set of possible actions that are performed on all the instances of the pattern

[17] The free Education edition of DB-MAIN is available at the following address: http://www.info.fundp.ac.be/libd, select "DB-MAIN CASE".

in the current schema. The right part allows the analyst to build linear transformation plans that can be saved and reused later.

The second assistant is more powerful, and therefore more complex. It is based on predicate-driven transformations following the syntax Σ(p) described in Section 5.4, and illustrated in Fig. 16. It allows non-linear transformation plans to be developed.

Part 2. Formal Aspects of Database Transformations

These sections which follow provide the bases for building a formal system in which GER transformations can be rigorously defined and such properties as semantics preservation can be studied.

9 The Extended Relational Model (ERM)

ERM is a variant of the N1NF relational model. It includes the concepts of domain, relation (schema and instance), attribute and constraints.

9.1 Domain

A domain is a named set of elements. It is declared by its name and the specification of the set of elements. A domain is *dynamic* if its set can change over time. Some *predefined basic domains* are provided, such as number, string or date. The model includes a special dynamic basic domain, called entities, whose structure is immaterial, but the goal of which could be to denote application domain entities. A *user-defined domain* is defined by an element set which is a subset of that of another domain. A relation is a valid domain. Any domain defined as a subset of the domain entities is an *entity domain*, and so forth transitively.

Example of user-defined domains

```
birth_Date:   date;
name:         string;
PERSON:       entities;
EMPLOYEE:     PERSON;
CONTACT:      address;
```

9.2 Relation and Attribute

According to the relational theory, a relation is a subset of the cartesian product of domains. An element of a relation is a tuple. A relation is described by its *schema*, that specifies the format and the constraints that its instances must satisfy. The current *instance* of a relation is the current set of tuples.

The schema of a relation comprises its name, a set of attributes and a set of constraints. An *attribute* has a name and is defined on a domain. It represents a participation of a domain in the relation. A domain can appear more than once, defining as many distinct attributes. An attribute defined on an entity domain is an *entity attribute*.

In general, the value of an attribute of a tuple is a subset of its domain. To specify the size of this subset, a cardinality property [i-j] is associated with each attribute A. It states the minimum and maximum numbers of domain values that are assigned to A in any tuple. If j = 1, A is single-valued otherwise it is multivalued. If i = 0, A is optional otherwise it is mandatory. The default cardinality property is [1-1].

Examples

```
address (     Street:          name,
              City:            name );
employee (    PId:             number,
              Name:            name,
              1st-name[0-1]:   name,
              Phone[1-5]:      phone,
              Contact:         address);
```

Interpretation: an employee has one (default [1-1], that is, *from 1 to 1*) personal ID, one name, from 0 to 1 first name, from 1 to 5 phone numbers, and one contact, which is made up of one street and one city.

If the concept of *address* is not considered important (for instance, it is not referred to elsewhere), the domain address could be specified in line as follows:

```
employee (…, Contact: (Street: name, City: Name));
```

In some situations, the specification of the domain will be ignored for simplicity. Consequently, the following notation will be allowed.

```
employee (PId, Name, 1st-name[0-1], Phone[1-5], Contact: address);
```

In particular, specially in formal declarations, if an attribute is given the name of its domain, we will use the following shorthand, where A is both the name of a domain and an attribute defined on it:

```
R(A,B,C)  ≡  R(A:A, B:B, C:C)
```

9.3 Non-set Attributes

By default, the value of an attribute is a *set* of domain values. Due to the generality of the GER, that is intended, among others, to describe logical and physical schemas, we need more poweful data structures, such as set, bag, list and array attributes:

```
R (A, B[0-5]set:number, C);   also defined as:   R (A, B[0-5]:number, C)
R (A, B[0-5]bag:number, C);
R (A, B[0-5]list:number, C);
R (A, B[0-5]array:number, C);
```

When the values in a list or in an array have to be *unique*, we write:

```
R (A, B[0-5]u-list:number, C);
R (A, B[0-5]u-array:number, C);
```

9.4 Constraints

ERM includes the **uniqueness** and **inclusion** constraints, as well as various dependencies, such as functional (**FD**) and multivalued (**MV**), of the standard relational model[18]. Candidate key {A,B,C} of R will be declared by the clause id(R): A,B,C. When possible, and where no ambiguity may arise, this specification can be replaced by continuously underlining the components of the key. Inclusion constraints between algebraic expressions are allowed.

Examples

1. R (A, B, C, D); id(R): A,B; *also defined as*: R (A, B, C, D)[19]
2. S(E, G, H); S[G,H] ⊆ R[A,B];
3. T(A, B, C); T[A] = A;

Example 1 declares a *candidate key* in both alternative syntaxes. Example 2 declares a *foreign key* through an inclusion constraint. Expression R[G,H] denotes the projection of the current instance of R on attributes (A,B). Example 3 expresses a domain constraint. Every element of domain A must appear as the value of attribute A of at least one tuple of the current instance of T.

In a N1NF structure, a local key can hold in a multivalued compound attribute. In the following example, we declare that, for each product tuple, the candidate key {Year} holds in each instance of Sales (no two sales the same year):

```
product (   ProNbr:          number,
            Description:     name,
            Sales[0-N]:      (Year: date, Volume: number));
```

The notation is extended as follows:

```
id(product.Sales): Year;
```

or by underlining the components:

```
product (ProNbr, Description, Sales[0-N]: (Year, Volume));
```

ERM includes a special form of **cardinality constraint**, through which we can state how many tuples of the current instance of a relation must/can share a common domain value.

---

[18] These constraints have been defined on 1NF models, and their generalization to N1NF models is far from trivial. Due to the limited scope of this paper, and without loss of generality, we will ignore the complexity of the constraint patterns of N1NF models.

[19] The graphical convention is as follows: the key of R(A,B,C) is {A,B} while R(A,B,C) has two keys {A} and {B}.

Considering the relation schema R(A,B,C) and an instance r of R,

$$\text{card}(R.A): [I\text{-}J],$$

is interpreted as[20]

$$\forall a \in A,\ I \leq |r(A=a)| \leq J$$

Examples

1. R (A, B, C); card(R.A): [0-5];
2. R (A, B, C); card(R.(B,C)): [1-3];

Example 1 declares that any value of domain A may not appear in more than 5 tuples of (any instance of) R. Example 2 shows a generalization of the constraint. It declares that any couple of values of domains B and C must appear in 1 to 3 tuples of (any instance of) R.

Note that candidate keys as well as the domain constraint T[A] = A are special cases of cardinality constraint. Note also that cardinality properties and cardinality constraints serve different purposes, and that none can replace the other one.

9.5 An ERM Schema Example

We are now able to propose a more comprehensive example of ERM schema.

- **domains**

  CUSTOMER: entities;
  VEHICLE: entities;
  CAR, BOAT: VEHICLE;
  Name: string;

- **relations**

  cust (CUSTOMER, CId: number, Name: name, Phone[0-3]: string),
  owns (owner: CUSTOMER, CAR);

- **constraints**

  VEHICLE = CAR ∪ BOAT;
  id(cust): CId;
  card(owns.owner): [0-5];
  owns(CAR) = CAR;

10 Formal Semantics of the GER

The mapping Σger>erm (Section 3.3) is fairly straighforward for most GER constructs. The inverse mapping is easy to derive as well. The main rules are presented in

[20] Expression r(A=a) denotes the set of tuples of r where A=a.

Fig. 24, and need little explanation, except for the representation of an entity type, since it seems to differ from the usual way one translates a conceptual schema into relational structures, as illustrated in Fig. 1 for example. First, let us recall that the goal of this section is not to produce relational databases, as discussed in Section. 6.2, but rather to give an operational model rigorous semantics.

An entity type E is merely represented by an entity domain, with name E, independently of any other feature, such as attributes, it may be concerned with.

When entity type E participates in relationship type (rel-type for short) R, with role r, its representation also appears as the domain of ERM attribute r of the relation R that expresses this rel-type (see rel-types of and export in Fig. 24).

Now, how to express the GER attributes of E? Through a special relation that aggregates each entity with its GER attribute values. The relation is given the conventional name desc-E, for *description of E*. This relation comprises an entity attribute, with name E, and defined on entity domain E. This attribute is a key of the relation. Then, for each GER attribute, it comprises an ERM attribute, with the same name and the same domain. Later on, we will see that, in some circumstances, this relation can include other entity domains.

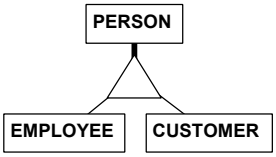
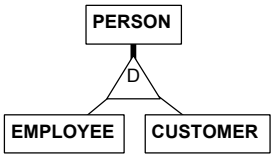
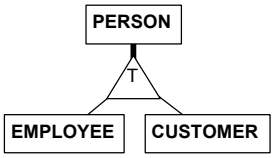
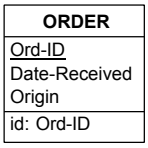
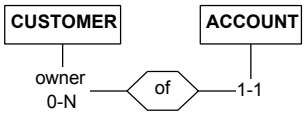
In this way, we can easily describe, beyond plain GER structures, an entity type without attributes, or without identifiers, or with complex constraint patterns.

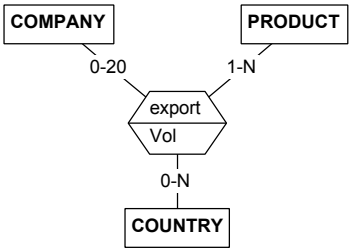
Note on the representation of functional relationship types

A rel-type is *functional* if it is binary, has no attributes and if at least one of its roles has cardinality [i-1]. Let us consider the functional rel-type of, between ACCOUNT and CUSTOMER, in Fig. 4, and recalled in Fig. 24. These three constructs translate in ERM as follows (note that the identifier of ACCOUNT has not been translated yet):

```
CUSTOMER, ACCOUNT: entities;
desc-CUSTOMER(CUSTOMER, …);
desc-ACCOUNT(ACCOUNT, Account-Nbr, Amount);
of(CUSTOMER, ACCOUNT);
desc-CUSTOMER[CUSTOMER] = CUSTOMER;
desc-ACCOUNT[ACCOUNT] = ACCOUNT;
of[ACCOUNT] = ACCOUNT;
```

This schema happens to meet the preconditions of the semantics-preserving *project-join* transformation that will be studied in Section 11.1. Its application yields the following equivalent, but simpler, schema, in which the relations desc-ACCOUNT and of have been joined:

GER constructs	ERM constructs
	PERSON: entities;
	PERSON: entities; EMPLOYEE: PERSON; CUSTOMER: PERSON;
	PERSON: entities; EMPLOYEE: PERSON; CUSTOMER: PERSON; EMPLOYEE ∩ CUSTOMER = ∅
	PERSON: entities; EMPLOYEE: PERSON; CUSTOMER: PERSON; EMPLOYEE ∪ CUSTOMER = PERSON
	ORDER: entities; desc-ORDER(ORDER, Ord-ID: number, Date-received: date, Origin: string); id(desc-ORDER): Ord-ID;
	of(owner: CUSTOMER, ACCOUNT); of[ACCOUNT] = ACCOUNT;

	export(COMPANY, PRODUCT, COUNTRY, Vol);
	export[PRODUCT] = PRODUCT;
	card(export.COMPANY): [0-20];

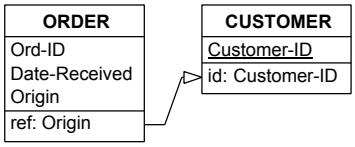
	desc-ORDER(ORDER, …, Origin: string);
	desc-CUSTOMER(CUSTOMER, Customer-ID);
	id(desc-CUSTOMER): Customer-ID;
	desc-ORDER[Origin]
	⊆ desc-CUSTOMER[Customer-ID];

Fig. 24. Main GER-to-ERM transformations (left to right) and their inverse (right to left)

CUSTOMER, ACCOUNT: entities;
desc-CUSTOMER(CUSTOMER, …);
desc-ACCOUNT'(ACCOUNT, Account-Nbr, Amount, Customer: CUSTOMER);
desc-CUSTOMER'[CUSTOMER] = CUSTOMER;
desc-ACCOUNT'[ACCOUNT] = ACCOUNT;

This form is quite interesting. Indeed, it allows us to specify, in a particularly simple and elegant way, complex constraints, such as hybrid identifiers, that is, identifiers that combine attributes and/or remote roles. Such an identifier is associated with entity type ACCOUNT in Fig. 4, the legend of which tells us that *the accounts of a customer have distinct Account numbers, which makes [ACCOUNT] a dependent or weak entity type*. Specifying this identifier is straightforward:

id(desc-ACCOUNT)': Customer, Account-Nbr

11 The ERM Transformations

In this section, we describe five important families of semantics-preserving parametric transformations that can be applied to ERM schemas. Basically, they are relational transformation and could be applied to any N1NF schema as well.

For each family, after a description of the principles, we specify the structural mapping **T**, through conditions **P** and **Q** (expressed in an intuitive way, through abstract structural patterns), if available, the description of useful variants, the signature of direct and inverse transformation, a discussion of their properties and an example. The **t** part will be ignored here. See [20] for a more detailed description of these transformations.

In the following descriptions, `U` is the set of attributes of relation `R`, while `I`, `J` and `K` denote subsets of `U`.

11.1 Project-join Transformations

Principle

A relation `R` in which a multivalued dependency (e.g., a FD) holds can be decomposed into smaller fragments according to this dependency [13].

Structural mapping

P	`R(U); {I, J, K}` is a partition of `U; I` $\rightarrow\!\!\rightarrow$ `J | K;`
Q	`R1(I J); R2(I K); R1[I]=R2[I];`

Variants

The project-join transformation can be particularized to relations in which I, J and/or K are made up of one attribute only, in which K is optional, in which K is multivalued, in which J is empty, and in which J and K are multivalued.

Signatures

direct : (R1,R2) ⟵ **PJ**(R,I,J)

reverse : R ⟵ **PJ**$^{-1}$(R1,R2,I)

Discussion

This transformation is the variant of the relational decomposition theorem mentioned in Section 4.3. It is therefore symmetrically reversible.

Example

Source schema
```
works(who:EMP,in:PROJ,for:DEPART)
works:who ⟶ for
```

Transformation
```
(works-in,works-for) ⟵ PJ(works,{who},{in})
```

Target schema
```
works-in(who:EMP,in:PROJ)
works-for(who:EMP,for:DEPART)
works-in[who] = works-for[who]
```

11.2 Denotation Transformation

Principle

The result of a query E defined by, say, an algebraic expression, and the schema of which comprises attributes AE, is explicitly represented in schema S with a denotational domain X. Bijective relation D acts as a dictionary for the elements of X. This operator is mainly technical and is used as a basis for the next transformation. It is trivially symmetrically reversible.

Structural mapping

P	schema S; algebraic expression **E** with schema SE(A_1,…,A_n)
Q	schema S; domain X; D(X,A_1,…,A_n); D[A_1,…,A_n] = **E**[A_1,…,A_n]; X appears in D only

Signatures

direct : (X,D,{A_1,…,A_n}) ⟵ **den**(S,**E**)

reverse : () ⟵ **den**$^{-1}$(X,D)

11.3 Extension Transformations

Principle

The projection of a relation R on a subset $\{I_1,\ldots,I_n\}$ of its attributes is explicitly represented by surrogate domain X. Bijective relation D acts as a dictionary for the elements of X. This domain replaces I in R, leading to relation T.

Structural mapping

P	R(U); {I,J} is a partition of U
Q	domain X; D(X,I); T(X,J); D[X] = T[X]; X appears in D and T only

Variants

When I = U, J is empty, so that the transformation degenerates into:

P	R(U);
Q	domain X; D(X,U); X appears in D only

If I comprises at least 2 attributes, it can be partitioned into subsets $\{I_1,..,I_m\}$. Considering the FD $D:X \longrightarrow I$, we can apply the project-join transformation to D according to this partition. Expressing the lost FD $D:I \longrightarrow X$ on the join of the fragments, we get the two **extension-decomposition** transformations (according to whether J is not empty or empty):

P	R(U); $\{I_1,..,I_m,J\}$ is a partition of U; m > 1
Q	$D_i(X,I_i)$; T(X,J); $D_i[X] = T[X]$; $i\in[1..m]$ $(*D_i, i\in[1..m]): I_1,..,I_m \longrightarrow X$; X appears in D_i and T only; $i\in[1..m]$

P	R(U); $\{I_1,..,I_m\}$ is a partition of U; m > 1
Q	$D_i(X,I_i)$; $D_i[X] = D_j[X]$; $i,j\in[1..m]$ $(*D_i, i\in[1..m]): I_1,..,I_m \longrightarrow X$; X appears in D_i only; $i\in[1..m]$

Signatures

Extension

direct : (X,D,T) ⟵ **ext**(R,I)

reverse : R ⟵ $\mathbf{ext}^{-1}$(X,D,T)

Extension decomposition

direct : $(X,\{D_1,D_2,..,D_m\},T) \longleftarrow$ **ext-dec**$(R,\{I_1,I_2,..,I_m\})$

reverse : $R \longleftarrow$ **ext-dec**$^{-1}(X,\{D_1,D_2,..,D_m\},T)$

For transformations where J is empty, parameter T is void.

Discussion

This family of operators is particularly powerful, since it allows us to generate most entity-generating and entity-removing transformations [17]. Based on the **den** and **PJ-1** transformations, it is symmetrically reversible. The role of the parameter I can be interpreted as follows: *the subset I of attributes of R seems to represent an outstanding concept which would deserve being described by a new surrogate domain X.*

Example of the extension transformation

Source schema
```
program(TEACHER,SUBJECT,DATE)
```

Transformation
```
(LECTURE,defined-as,program) ⟵
ext(program,{TEACHER,SUBJECT})
```

Target schema
```
domain LECTURE
program(LECTURE,DATE)
defined-as(LECTURE,TEACHER,SUBJECT)
defined-as[LECTURE] = program[LECTURE]
```

11.4 Composition Transformations

Principle

A relation S is replaced by its composition T with another relation R.

Structural mapping

P	R(I K); S(K L); S[K] ⊆ R[K]; I,K,L not empty;
Q	R(I K); T(I L); T[I] ⊆ R[I]; R*T: K →→ L|I

Variants

The transformation simplifies when R is bijective:

P	R(I K); S(K L); S[K] ⊆ R[K]; I,K,L not empty;
Q	R(I K); T(I L); T[I] ⊆ R[I];

The latter form generalizes to N-ary relations:

P	R(<u>I K</u> J); S(K L); S[K] ⊆ R[K]; I,J,K,L not empty;
Q	R(<u>I K</u> J); T(I L); T[I] ⊆ R[I]

Signatures (simple form)

direct : T ⟵ **comp**(R,S,K)
reverse : S ⟵ **comp**$^{-1}$(R,T,I)

Signatures (N-ary form)

direct : T ⟵ **comp**(R,S,K,I)
reverse : S ⟵ **comp**$^{-1}$(R,T,I,K)

Discussion

These operators derive from transformations **PJ** and **PJ**$^{-1}$. Therefore they are symmetrically reversible. In the bijective variants, the transformation is symmetrical and can be seen as substituting in S a key I of R for the key K.

Example

Source schema
manages(<u>MANAGER,DEPART</u>)
works-in(<u>EMPLOYEE</u>,DEPART)
works-in[DEPART] ⊆ manages[DEPART]

Transformation
works-for ⟵
comp(manages,works-in,{DEPART})

Target schema
manages(<u>MANAGER,DEPART</u>)
works-for(<u>EMPLOYEE</u>,MANAGER)
works-for[MANAGER] ⊆ manages[MANAGER]

11.5 Nest-unnest Transformations

Principle

A N1NF relation R that comprises a multivalued attribute B is replaced by S, its equivalent 1NF version [43] [31].

Structural mapping

P	R(<u>I</u>,B[1-N]);
Q	S(<u>I,B</u>);

Variants

The cardinality of attribute B prohibits empty sets (otherwise values of I are lost), which can be too strong a precondition. Hence the following variant, in which the

tuples of R with an empty B set can be rebuilt from the elements of the evaluation of **E** that do not appear in S:

P	R(I,B[0-N]); R[I] = **E**; where **E** is any algebraic expression over the database schema
Q	S(I,B); S[I] ⊆ **E**;

If B is a compound but single-valued attribute, this operator degenerates into a disaggregation transformation as follows, where K is a set of attributes:

P	R(I,B(K));
Q	S(I,K);

Signatures

direct : S ⟵ **unnest**(R,B)

reverse : R ⟵ **unnest**$^{-1}$(S,B)

Discussion

Unnest, together with its inverse *nest*, are the main algebraic operators specific to N1NF relational models. This version of **unnest** is symmetrically reversible. Indeed, R meets the following criterion of reversibility (see [10] for instance): considering the relation R(A,B[0-N],C), the application of the *unnest* relational operator on B is (symmetrically) reversible *iff*:

- no tuple of R has an empty B value (as if the cardinality property of B actually was [1-N]),
- B is functionally (possibly non minimally) dependent on the set of all the other attributes of R.

Examples

Source schema contacts(EMPLOYEE,PHONE[1-N])

Transformation contact ⟵ **unnest**(contacts,PHONE)

Target schema contact(EMPLOYEE,PHONE)

Source schema descr(EMPLOYEE,CHILD[0-N])
descr[EMPLOYEE] = EMPLOYEE

Transformation children ⟵ **unnest**(descr,CHILD)

Target schema children(EMPLOYEE,CHILD)

Note in this example the instance "descr[EMPLOYEE] = EMPLOYEE" of the pattern "R[I] = **E**".

12 Analyzing and Generating GER Transformations

12.1 Analyzing GER Transformations

The issue is to prove that a known, but possibly ill-defined, practical transformation is *correct* and *complete* as far as semantics preservation is concerned. In this context, we will revisit the three transformations that we have informally used in the introductory example of Fig. 1, and that also are the most popular, notably in database logical design. Due to space limit, only the main patterns will be discussed. For any variant of the source schema, such as those that are suggested below, the reader is invited to examine the ERM expression and to infer the actual resulting schema. For example, in the transformation of attribute A2 into an entity type, no hypothesis is made on the participation of A2 in an identifier of A. If this is the case, the ERM expression clearly shows how to deal with this pattern, based on the dependency theory[21]. This is left as an exercise.

12.2 Transforming an Attribute into an Entity Type

In Fig. 1, this transformation was applied to attribute Author of BOOK, leading to entity type AUTHOR. Its abstract GER pattern is as follows.

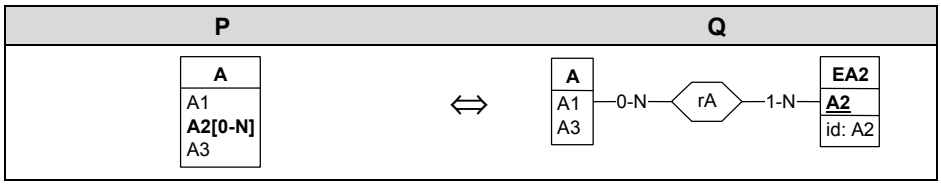


Fig. 25. Transforming an attribute into an entity type

Variants. The reader is invited to examine the following extensions: A2 is single-valued; A2 is an identifying attribute for A; A2 is a component of an identifier of A; A is a compound attribute; the cardinality property is [0-5] or [1-5]; A2 is a set of attributes of A.

Signatures

direct : `(EA2,rA)` ⟵ **`att-to-et`**`(A,A2)`

reverse : `A2` ⟵ **`att-to-et`**$^{-1}$`(EA2)`

[21] More precisely the rules that govern the propagation of FD in the projection, the join and the selection.

Analysis

We express the source schema (left) in ERM, then we extract and flatten the multivalued attribute:

```
A: entities;
desc-A(A,A1,A2[0-N],A3);
desc-A[A]=A;
```

⇳ (desc-A',R) ⟵ **PJ**(desc-A,{A},{A2})

```
A: entities;
desc-A'(A,A1,A3);
R(A,A2[1-N]);
desc-A'[A]=A;
```

⇳ R' ⟵ **unnest**(R,A2)

```
A: entities;
desc-A'(A,A1,A3);
R'(A,A2);
desc-A'[A]=A;
```

Now, we define a new entity domain EA2 based on attribute A2 of R':

⇳ (EA2,{desc-EA2},rA) ⟵ **ext**(R',{A2})

```
A,EA2: entities;
desc-A'(A,A1,A3);
desc-EA2(EA2,A2);
rA(A,EA2);
desc-A'[A]=A;
desc-EA2[EA2]=rA[EA2]=EA2;
```

Interpreting this schema in the GER gives the expected target schema (right). We conclude that **att-to-et** is an SR-transformation.

12.3 Transforming a Relationship Type into an Entity Type

In the illustration of Fig. 1, we transformed relationship type write into entity type WRITE. Here is a generalization of this operator for N-ary relationship types, that can also have attributes (Fig. 26).

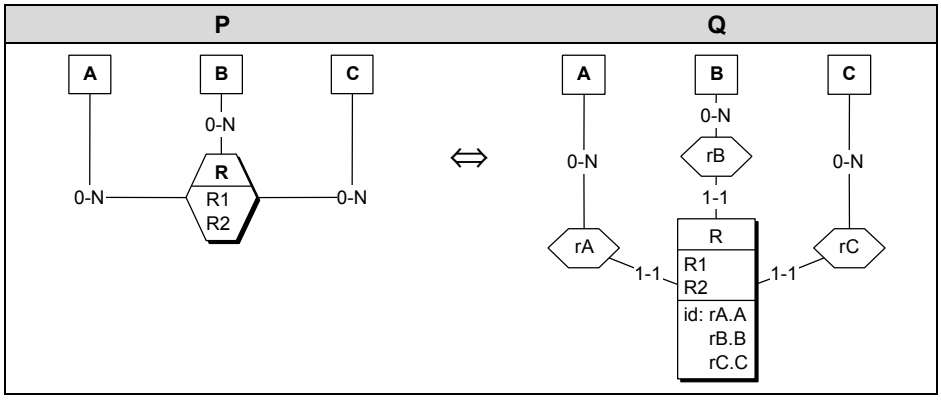


Fig. 26. Transforming a relationship type into an entity type

Variants. The roles of R have cardinality constraints other than [0-N]; R is binary; one (or more) of the roles of R has cardinality [0-1]; R has one (or more) explicit identifier[22].

Signatures

direct : `(R,{(A,rA),(B,rB),(C,rC)}) ⟵ rt-to-et(R)`

reverse : `R ⟵ rt-to-et⁻¹(R)`

Analysis

We express the source schema (left) in ERM, then we represent the set of roles by the new entity domain R:

```
A,B,C: entities;
R(A,B,C,R1,R2);
desc-A[A]=A;
```

⇕ `(R,{rA,rB,rC},desc-R) ⟵ ext-dec(R,{{A},{B},{C}})`

```
A,B,C,R: entities;
rA(R,A); rB(R,B); rC(R,C);
desc-R(R,R1,R2);
rA*rB*rC: A,B,C ⟶ R;
rA[R]=rB[R]=rC[R]=desc-R[R]=R;
```

Interpreting this schema in the GER gives the expected target schema (right). We conclude that **rt-to-et** is an SR-transformation.

[22] The default (not necessarily minimal) identifier of a relationship type is made up of the set of its roles.

12.4 Transforming a Binary Relationship Type into an Attribute

In Fig. 1, we transformed all the one-to-many relationship types into attributes, then we declared them foreign keys.

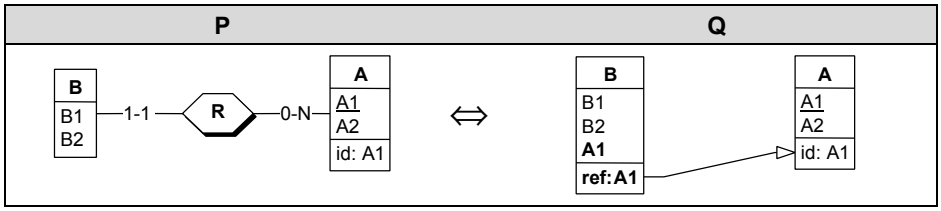


Fig. 27. Transforming a relationship type into an attribute (foreign key)

Variants. R is optional for B ([1-1] replaced by [0-1]); R is many-to-many ([1-1] replaced with [0-N]); the identifier of A is made up of more than one attribute; R is functional from A to B ([0-N] replaced by [0-1]); R is bijective; R is mandatory for A ([0-N] replaced by [1-N]); R.A appears in an identifier of B.

Signatures

direct : `{A1} ⟵ rt-to-att(R.B)`

reverse : `R ⟵ rt-to-att`$^{-1}$`(B,{A1},A)`

Analysis

We express the source schema (left) in ERM, then we apply the composition transformation:

```
A,B: entities;
desc-A(A,A1,A2); desc-B(B,B1,B2); R(A,B);
R[B]=B; desc-A[A]=A; desc-B[B]=B;
```

⇕ `R' ⟵ comp(desc-of-A,R,{A},{A1})`

```
A,B: entities;
desc-A(A,A1,A2); desc-B(B,B1,B2); R'(A1,B);
desc-B[B]=R'[B]=B; desc-A[A]=A;
R'[A1] ⊆ desc-A[A1];
```

⇕ `desc-B' ⟵ PJ`$^{-1}$`(desc-B,R',B)`

```
A,B: entities;
desc-A(A,A1,A2); desc-B'(B,B1,B2,A1);
desc-A[A]=A; desc-B'[B]=B;
desc-B'[A1] ⊆ desc-A[A1];
```

Interpreting the latter schema in the GER gives the expected target schema (right). We conclude that **rt-to-att** is an SR-transformation.

12.5 Generating GER Transformations

This process consists in exploiting the parametric nature of most ERM transformations to discover new practical GER transformations. This problem is open, but we can illustrate it through a more in-depth examination of the extension-decomposition transformation.

Let us consider the transformation depicted in the Fig. 26. Its analysis is based on the ERM ext-dec transformation of the ERM relation R(A,B,C,R1,R2) that models the relationship type R.

The GER rt-to-et transformation we have developed was obtained by choosing, in the ERM ext-dec transformation, the parameter I to be {A,B,C}. In fact, I is any non empty subset of the attributes of relation R. For instance, I can be any of the following subsets, that will generate 31 different equivalent target schemas:

{A}, {A,B}, {A,B,C}, {R1}, {R1,R2}, {A,R1}, {A,R1,R2}, {A,B,R1}, {A,B,R1,R2}, {A,B,C,R1}, {A,B,C,R1,R2}, and all the similar patterns obtained by permutation within {A,B,C} and {R1,R2}.

The reader is invited to prove the correctness of the transformation of Fig. 28 following the reasoning of Section 12.3.

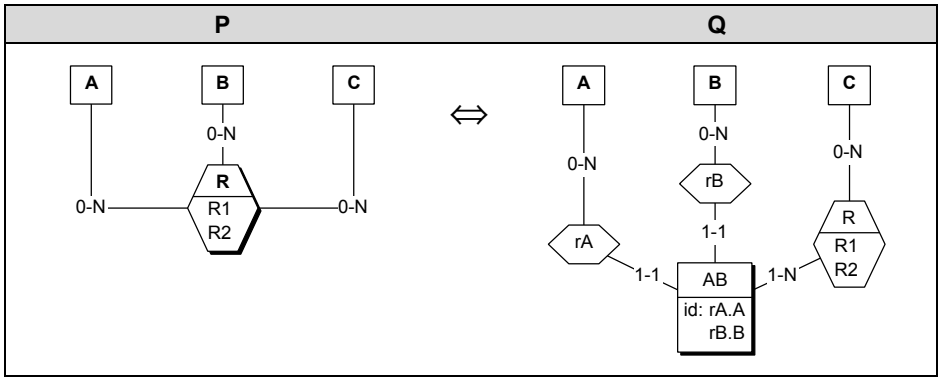


Fig. 28. An unusual transformation deriving from the ext-dec transformation

13 Conclusions and Perspectives

Database engineering intrinsically has been model-driven for more than three decades. Designing, normalizing, merging, optimizing data structures can be performed at an abstraction level that is, to a large extent, platform independent.

The transformational approach enriches this framework considerably, since it opens the way to more structured and more reliable engineering processes. This paper shows that such an approach brings several essential benefits.

- Being formal, it can be used to study rigorously basic properties such as semantics preservation, that states how the operators preserve the information contents of the schemas;
- To be fruitful, and to avoid combinatorial explosion, a pivot model, with which we associate a relational semantics, has proved necessary;
- From the pedagogical view point, this approach provides a disciplined and reliable way to conduct important processes such as logical design, which many students too often tend to consider as some kind of *magic*;
- Developing CASE tools based on the transformational approach leads to more reliable products, notably as far as generation completeness is concerned;
- A transformational approach based on a pivot model is by construction scalable; introducing a new model M involves the development of components independent of the existing models.

Several problems still are to be addressed, of which we mention a sample.

- How to integrate transformational database engineering into emerging MDE framework(s)?
- How to cope with the other aspects of data structures, in particular how do integrity constraints propagate?
- How can data structure transformations be propagated to the other components of the information system, notably the data (data conversion), the human/computer interfaces and the programs?

14 References and Resources

1. Alves, T.L., Silva, P.F., Visser, J., Oliveira, J.N., Strategic Term Rewriting and Its Application to a Vdm-SL to SQL Conversion, in *Proc. FM 2005*, LNCS, No 3582, Springer-Verlag. (2005) 399-414
2. Baader, F., Horrocks, I., and Sattler, U. Description logics. In Staab, S. and Studer, R. (Ed.), *Handbook on Ontologies*, International Handbooks on Information Systems, pages 3-28. Springer, (2004).
3. Balzer, R. Transformational implementation : An example. *IEEE TSE*, Vol. SE-7(1). (1981)
4. Batini, C., Ceri, S., & Navathe, S., B. *Conceptual Database Design*, Benjamin/Cummings. (1992)
5. Batini, C., Di Battista, G., Santucci, G. Structuring Primitives for a Dictionary of Entity Relationship Data Schemas, *IEEE TSE*, Vol. 19, No. 4. (1993)
6. Bolois, G., & Robillard, P. Transformations in Reengineering Techniques. *Proc. of the 4th Reengineering Forum "Reengineering in Practice"*, Victoria, Canada. (1994)
7. Boyd, M., McBrien. Towards a Semi-Automated Approach to Intermodel Transformation, In *Proceedings of EMMSAD'04,Volume 1, CAiSE Workshop Proceedings*, Riga Technical University. (2004) 175-188
8. Casanova, M., A., Amaral De Sa. Mapping uninterpreted Schemes into Entity-Relationship diagrams : two applications to conceptual schema design. *IBM J. Res. & Develop.*, 28(1). (1984)

9. Clève, A., Henrard, J., Hainaut, J-L. Co-transformations in Information System Reengineering, in *Proc. of WCRE'04/ATEM-04*, (2004)
10. Darwen, H., Date, C., J. Relation-valued Attributes, in Date, C., J., Darwen, H., *Relational Database Writings* 1989-1991, Addison-Wesley (1993)
11. D'Atri, A., & Sacca, D. Equivalence and Mapping of Database Schemes, *Proc. 10th VLDB conf.*, Singapore. (1984)
12. Estiévenart, F., François, A., Henrard, J., Hainaut, J-L. Web Site Engineering. *Proc. of the 5th International Workshop on Web Site Evolution*, Amsterdam, Sept. 2003, IEEE CS Press. (2003)
13. Fagin, R. Multivalued dependencies and a new normal form for relational databases, *ACM TODS*, 2(3). (1977)
14. Fikas, S., F. Automating the transformational development of software, *IEEE TSE*, Vol. SE-11. (1985)
15. Hainaut, J-L. Theoretical and practical tools for database design, in *Proc. of the Very Large Databases Conf.*, pp. 216-224, September, IEEE Computer Society Press. (1981)
16. Hainaut, J.-L. A Generic Entity-Relationship Model. *Proc. of the IFIP WG 8.1 Conf. on Information System Concepts: an in-depth analysis*, North-Holland. (1989)
17. Hainaut, J-L. Entity-generating Schema Transformations for Entity-Relationship Models, in *Proc. of the 10th Entity-Relationship Approach*, San Mateo (CA), 1991, North-Holland. (1992)
18. Hainaut, J-L., Chandelon M., Tonneau C., & Joris M. (1993). Contribution to a Theory of Database Reverse Engineering. *Proc. of the IEEE Working Conf. on Reverse Engineering*, Baltimore, May 1993, IEEE Computer Society Press.
19. Hainaut, J-L, Chandelon M., Tonneau C., Joris M. Transformational techniques for database reverse engineering. *Proc. of the 12th Int. Conf. on ER Approach*, Arlington-Dallas, ER Institute (and LNCS Springer-Verlag in 1994). (1993)
20. Hainaut, J-L. *Transformation-based database engineering*. Tutorial notes, *VLDB'95*, Zürich, Switzerland, (1995) (available at http://www.info.fundp.ac.be/libd).
21. Hainaut, J-L. Specification preservation in schema transformations - application to semantics and statistics, *Data & Knowledge Engineering*, 11(1). (1996)
22. Hainaut, J-L., Henrard, J., Hick, J-M., Roland, D., Englebert, V. Database Design Recovery, in *Proc. of the 8th Conf. on Advanced Information Systems Engineering* (CAiSE•96), Springer-Verlag (1996)
23. Hainaut, J.-L., Hick, J.-M., Englebert, V., Henrard, J., Roland, D. Understanding implementations of IS-A Relations, in *Proc. of the conference on the ER Approach*, Cottbus, Oct. 1996, LNCS, Springer-Verlag (1996).
24. Hainaut, J-L. Transformation-based Database Engineering. In: [47]. (2005) 1–28
25. Halpin, T., A., & Proper, H., A. Database schema transformation and optimization. *Proc. of the 14th Int. Conf. on ER/OO Modelling* (ERA). (1995)
26. Henrard, J., Hick, J-M. Thiran, Ph., Hainaut, J-L. Strategies for Data Reengineering, in *Proc. of WCRE'02*, IEEE Computer Society Press. (2002)
27. Hick, J-M., Hainaut, J-L. Strategy for Database Application Evolution: the DB-MAIN Approach, in *Proc. ER'2003 conference*, Chicago, Oct. 2003, LNCS Springer-Verlag. (2003)
28. Jajodia, S., Ng, P., A., & Springsteel, F., N. The problem of Equivalence for Entity-Relationship Diagrams, *IEEE Trans. on Soft. Eng.*, SE-9(5). (1983)
29. Kobayashi, I. Losslessness and Semantic Correctness of Database Schema Transformation : another look of Schema Equivalence, *Information Systems*, 11(1). (1986) 41-59
30. Lämmel, R. Coupled Software Transformations (Extended Abstract), In *Proc. First International Workshop on Software Evolution Transformations (SET 2004)*. (2004) [http://banff.cs.queensu.ca/set2004/set2004_proceedings_acrobat4.pdf]

31. Levene, M. *The Nested Universal Relation Database Model*, LNCS 595, Springer-Verlag. (1992)
32. Lien, Y., E. On the equivalence of database models, *JACM*, 29(2). (1982)
33. Ling, T., W. External schemas of Entity-Relationship based DBMS, in *Proc. of Entity-Relationship Approach : a Bridge to the User*, North-Holland. (1989)
34. McBrien P., & Poulovassilis, A. Data integration by bi-directional schema transformation rules, *Proc 19th International Conference on Data Engineering* (ICDE'03), IEEE Computer Society Press. (2003)
35. Motro, Superviews: Virtual integration of Multiple Databases, *IEEE Trans. on Soft. Eng*. SE-13, 7, (1987)
36. Navathe, S., B. Schema Analysis for Database Restructuring, *ACM TODS*, 5(2), June 1980. (1980)
37. Partsch, H., & Steinbrüggen, R. Program Transformation Systems. *Computing Surveys*, 15(3). (1983)
38. Poole, J. Model-Driven Architecture : Vision, Standards And Emerging Technologies. in *Proc. of ECOOP 2001*, Workshop on Metamodeling and Adaptive Object Models, (2001)
39. Rauh, O., & Stickel, E. Standard Transformations for the Normalization of ER Schemata. *Proc. of the CAiSE•95 Conf.*, Jyväskylä, Finland, LNCS, Springer-Verlag. (1995)
40. Roland, D. *Database engineering process modelling*, PHD Thesis, University of Namur. *http://www.info.fundp.ac.be/~dbm/publication/2003/these-dro.pdf* (2003)
41. Rosenthal, A., & Reiner, D. Theoretically sound transformations for practical database design. *Proc. of Entity-Relationship Approach*. (1988)
42. Rosenthal, & A., Reiner, D. Tools and Transformations - Rigourous and Otherwise - for Practical Database Design, *ACM TODS*, 19(2). (1994)
43. Schek, H-J., Scholl, M., H. () The relational model with relation-valued attributes, *Information Systems*, 11. (1986) 137-147
44. Thalheim, B. *Entity-Relationship Modeling: Foundation of Database Technology*. Springer-Verlag, (2000)
45. Thiran, Ph., Hainaut, J-L. Wrapper Development for Legacy Data Reuse. *Proc. of WCRE'01*, IEEE Computer Society Press. (2001)
46. Thiran, Ph., Estiévenart, F., Hainaut, J-L., Houben, G-J, A Generic Framework for Extracting XML Data from Legacy Databases, in *Journal of Web Engineering*, Rinton Press, (2005)
47. van Bommel, P. (Ed.). *Transformation of Knowledge, Information and Data: Theory and Applications*, Information Science Publ., Hershey. (2005)
48. van Griethuysen, J.J., (Ed.). Concepts and Terminology for the Conceptual Schema and the Information Base. Publ. nr. ISO/TC97/SC5-N695. (1982)