Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 65 (2008) 419–441
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

MeDEA: A database evolution architecture with traceability ☆

Eladio Domínguez [a], Jorge Lloret [a], Ángel L. Rubio [b,*], María A. Zapata [a]

[a] Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, E-50009 Zaragoza, Spain
[b] Dpto. de Matemáticas y Computación, Universidad de La Rioja, E-26004 La Rioja, Spain

Received 20 July 2007; accepted 17 December 2007
Available online 28 December 2007

Abstract

One of the most important challenges that software engineers (designers, developers) still have to face in their everyday work is the evolution of working database systems. As a step for the solution of this problem in this paper we propose MeDEA, which stands for Metamodel-based Database Evolution Architecture. MeDEA is a generic evolution architecture that allows us to maintain the traceability between the different artifacts involved in any database development process. MeDEA is generic in the sense that it is independent of the particular modeling techniques being used. In order to achieve this, a metamodeling approach has been followed for the development of MeDEA. The other basic characteristic of the architecture is the inclusion of a specific component devoted to storing the translation of conceptual schemas to logical ones. This component, which is one of the most noteworthy contributions of our approach, enables any modification (evolution) realized on a conceptual schema to be traced to the corresponding logical schema, without having to regenerate this schema from scratch, and furthermore to be propagated to the physical and extensional levels.

Keywords: Conceptual modeling; Database schemas evolution and maintenance

1. Introduction

Nowadays software developers have to face incessant technological challenges to achieve more efficient systems. Although these new technologies could be used when systems are built from scratch, more frequently they must be applied where existing systems have to be adapted, evolved or integrated with others. All aspects of systems development are influenced by this situation, and this is particularly true for one of the cornerstones of any current system: database systems and applications. On many occasions, existing databases must be adapted for several reasons (i.e. changes in business procedure, need of open-environments – such as

☆ This work has been partially supported by DGI, project TIN2005-05534, by the Ministry of Industry, Tourism and Commerce, project SPOCS (FIT-340001-2007-4), by the Government of La Rioja, project ANGI2005/19, by the Government of Aragon and by the European Social Fund.
* Corresponding author.
E-mail addresses: noesis@unizar.es (E. Domínguez), jlloret@unizar.es (J. Lloret), arubio@unirioja.es (Á.L. Rubio), mazapata@unizar.es (M.A. Zapata).

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2007.12.001

Internet-integration). For this reason, a framework allowing these adaptations to be realized without jeopardizing data or schema integrity would be a highly valuable tool.

In this paper we propose MeDEA, a Metamodel-based Database Evolution Architecture for managing evolution when all the components of the database (conceptual schema, logical schema and extension) are available. This architecture has been partially described in previous papers [7–10], but this is the first time it is explained in its entirety and in a general way independently of the specific application context. MeDEA provides a general framework which makes it feasible to handle changes within the different phases of the database life cycle, from early design stages to exploitation and maintenance, so that consistency among different database design levels is kept.

In order to do this, different kinds of models, at different levels of abstraction, must be managed and changes performed in a model must be automatically propagated to the others. One way of tackling this issue is to ensure the traceability of the translation process between levels. However, there is no agreement about which artifacts and mechanisms are needed for assuring traceability [29,36].

The way in which we propose to achieve traceability is making use of a specific translation component in which information related with the translation process is explicitly stored. The knowledge stored in this component avoids having to apply the translation algorithm from scratch when an evolution is performed. The idea of using an explicit component to store the elements related with the translation process and the way in which this knowledge is used during the database evolution process represent a significant contribution of our work. Furthermore, within our proposal, when an evolution process is carried out, the data are also modified to accommodate them to the new model. This fact allows us to achieve the 'incremental consistency' proposed as a desirable feature of transformation frameworks such as MDA [18].

A metamodeling approach [12,22] has been followed for the definition of MeDEA. Within this setting, three metamodels are considered which capture the conceptual, logical and translation modeling knowledge. We have chosen this approach because it allows modeling knowledge to be represented and because it has been proven that it facilitates the definition of data model translations [22]. The adoption of a metamodeling approach brings to our architecture the property of being independent of any particular modeling technique. However, in order to clarify the way the architecture works we will be considering a particular example, and for this reason some specific techniques must be established. For this example, we have chosen two widely accepted standard modeling languages: the Entity-Relationship Model at the conceptual level and the relational model at the logical level. So as to show a particular application of MeDEA, we present a current Oracle implementation. It should be noted that we have chosen these specific models and implementation only with the aim of illustrating our architecture. However, MeDEA is of general applicability and therefore can also be applied to other approaches (UML, object-oriented models, XML, etc.). For instance, in [8] we have applied MeDEA to the case of UML at the conceptual level and XML at the logical level.

The paper is organized as follows. In the following section the database evolution issue is described. Section 3 is devoted to presenting MeDEA, explaining its structure and its way of working. In order to clarify the way our architecture works, a specific case study is expounded in Section 4. Finally, we put forward the conclusions and outline future work.

2. The database evolution issue

There is a consensus about the process to be followed in order to develop a database system. Firstly, the application requirements are analyzed giving rise to a conceptual database schema S_C. This schema is represented by means of a modeling formalism such as (Extended) Entity-Relationship or UML. Secondly, the conceptual schema is translated into a logical database schema S_L, normally using a relational schema or an object-relational schema. Finally, the logical schema is implemented by means of a DBMS and the database is then populated to create a consistent database state σ (see left branch of Fig. 1a).

Once the database system is developed, or even during the development process itself, various reasons (changes in the environment requirements [15,33], detection of deficiencies in the performance [17], decision to migrate to a new platform [28], etc.) can require modifications to be made to the database. These changes can be managed preserving older versions of the schemas (schema versioning according to [30]) or not (schema

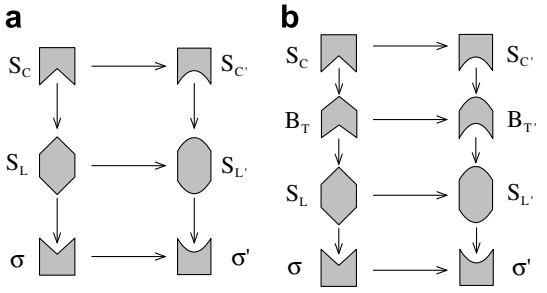

Fig. 1. Standard and enhanced forward database evolution proposals.

evolution [30]). In this paper, we tackle the database evolution issue using a schema evolution approach, so that we consider that only one version of the database is maintained.

The changes to be performed can imply modifications within different elements of the development process: conceptual schema S_C, logical schema S_L or database state σ. In any case, these modifications must be propagated to the rest of the artifacts. Depending on the case, a forward or a backward database maintenance strategy must be followed [16]. We are going to suppose that the changes are performed in the conceptual schema so that a forward strategy is followed. The application of our proposal in a context in which forward and backward strategies are considered remains an ongoing project.

Summing up, the standard forward evolution issue (see Fig. 1a) can be stated in a general way rephrasing the ideas explained in [33] as follows. For various reasons the conceptual schema S_C is modified generating a new conceptual schema S'_C. These changes must be automatically propagated down to the logical schema and its population. That is, the logical schema S_L has to be modified in order to generate a new logical representation S'_L and the database state σ has to be mapped into a new database state $\sigma\prime$ consistent with S'_L.

The changes to be performed in the conceptual database schema in order to carry out the desired evolution can be expressed by means of schema transformations [17]. A schema transformation accomplishes modifications in the structure of the database and maps the population of the source schema into an allowable population of the resultant schema [4,17].

Within this context the challenge is to keep the consistency between the different artifacts when a conceptual database evolution occurs. In order to do this, our proposal is to store knowledge about the translation process explicitly in a component, which we will call *translation base*, in the same way that the conceptual, logical and extensional information is stored, giving rise to an enhanced forward database evolution proposal (see Fig. 1b). The aim of this component is to store explicit information about the way in which specific conceptual elements are translated into logical ones.

We think that the consideration, within the database evolution issue, of our proposal of translation base brings several advantages. On the one hand, this component ensures the traceability of the translation process, which is established as a desirable feature of transformations [18] and included as such, for instance, in the QVT request for proposals [27]. Besides, this component allows the extensional changes to be analyzed in terms of conceptual changes so that, for example, conceptual notions such as relationship or specialization can be involved within logical evolution issues, even though logical models normally lack of these concepts [9].

Finally, the knowledge stored in the translation base avoids having to recalculate the logical elements that result from the conceptual elements that have not been modified, it being unnecessary to apply once again the translation algorithm from scratch. This is important not only for avoiding waste of computational resources, but also because the DBA or data designer has to make decisions only in relation to the modified elements.

3. MeDEA: a database evolution architecture

As an application of the database evolution approach described in the previous section, in this paper we propose MeDEA as a database evolution architecture. MeDEA aims at providing a general framework which makes it feasible to manage database evolution following a forward maintenance strategy. The more

noteworthy feature of this architecture is that it includes explicitly a component for the storage of information related with the translation process.

In order to explain the architecture we proceed in the following way. Firstly we will describe the different components that shape MeDEA (Section 3.1), and next we will present the way in which it comes into play (Section 3.2).

3.1. The structure of MeDEA

The architecture is configured making use of a structural artifact that consists of three components: a schema, an information base and a processor [14]. The schema defines all the knowledge relevant to the system, the base describes the specific objects perceived in the Universe of Discourse, and the processor receives messages reporting the occurrence of events in the environment. The inspiration for the way in which we bring into play this artifact came from [20]. In particular, within our architecture, this structural artifact is applied four times in order to store, respectively, the conceptual modeling knowledge, the translation process, the logical modeling knowledge and the extension. In the three former cases, given the nature of the stored information, we decided to follow a metamodeling approach, whereas a modeling approach was followed with regard to the extensional information. The resulting components as well as the way in which they are related appear in Fig. 2. The name of each one of these components has been adapted in order to capture the type of knowledge that they store.

It must be noted that three different abstraction levels are involved in the architecture. On the one hand, the (meta-)schemas of the three former components are situated at the most abstract level (metamodel layer according to [35]) and, on the other hand, the information base which stores the population of the database is situated at the least abstract level (user data layer [35]). All the other elements are situated at the model layer [35].

The *conceptual component* captures machine-independent knowledge of the real world. For instance, in the case of database evolution, this component could deal with entity-relationship schemas or with UML class diagrams modeling the domain. The *logical component* captures tool–independent knowledge describing the data structures in an abstract way. In database evolution, this component could deal with schemas from the (object-)relational model, as for instance by means of standard SQL, or with XML schemas. The *extensional component* captures tool dependent knowledge using the implementation language. In databases, it could deal with the specific database in question, populated with data, and expressed in the SQL of the DBMS of choice or in textual XML documents conforming to an XML schema.

Let us note that, in Fig. 2, the logical database schema is surrounded with both the information base and schema symbols. This is because the logical database schema can be seen as the information base of the logical component (as we have explained) or as the schema of the extensional component. For this reason two

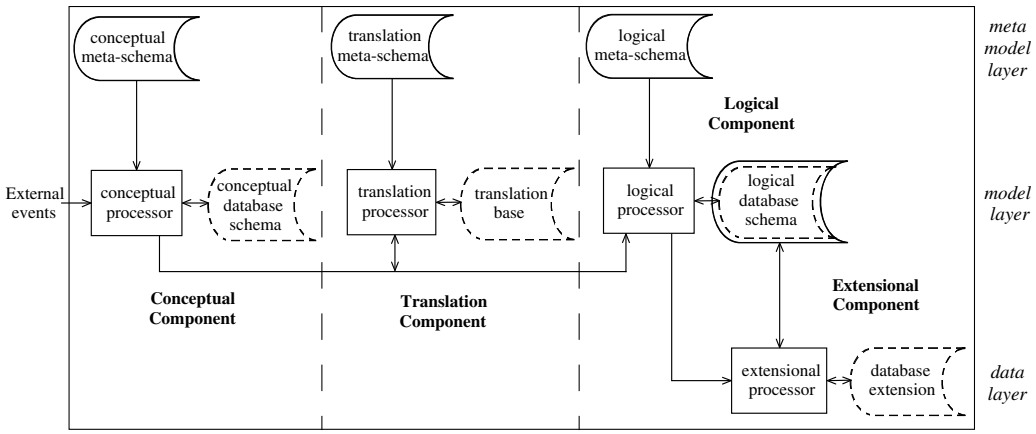

Fig. 2. Architecture for database evolution.

different elements of our architecture store the same information. This fact obliges us to define some rules, called *correspondence rules* (in the same sense as in [20]). These rules govern the correspondence between the elements of each one of the two components (see Section 3.2).

A more detailed description of the translation component is worthwhile as this is the novelty of our proposal. The aim of the *translation component* is to store all the information necessary to enable any change performed in the conceptual database schema to be automatically reflected in the logical schema. In order to achieve this goal, we propose several changes to the traditional way in which conceptual schemas are translated into logical schemas.

Usually this transformation process consists of applying a set of transformation rules to a conceptual schema in order to obtain a logical one. In the literature, there is a multitude of sources where sets of transformation rules are presented (see [13,25]). However, within MeDEA the transformation rules are defined in a particular way giving rise to a different *translation algorithm*. When this algorithm is applied to a model, it produces not only another model (as usual) but also a set of *elementary translations* stored in the translation component.

Specifically, the notion of transformation rule is enriched giving rise to a notion we call *translation rule*. The idea is that a translation rule not only specifies how the conceptual elements are translated into logical elements but also the process to be followed to store in the translation component the trace of the translation. Our proposal is that this trace gathers the applied rule together with its effects on the conceptual and logical schemas. In order to store the effects of each rule we introduce the notion of *elementary translation* defined as the smallest piece of information reflecting the correspondence between a conceptual element and a logical one. In this way the effect of the application of a rule is stored by means of a set of elementary translations.

The notions introduced above allow us to propose a translation metamodel pattern which appears in Fig. 3. This pattern shows that the translation component stores the translation rules applied to the conceptual schema together with the set of elementary translations describing which conceptual elements have been translated into which logical ones. This pattern has several variation points which have been specified by using the stereotype "variationPoint". This stereotype is proposed in [3] for modeling variability as part of a standard-conform extension to UML for developing generic models.

The variation points of this pattern must be determined when the architecture is applied to a specific evolution setting. In particular, the specific conceptual and logical elements of the conceptual and logical metamodels must be fixed as well as the types of translation rules and elementary translations considered (see Section 4).

3.2. The way of working of MeDEA

As we have explained previously, in order to develop a database system, firstly a conceptual database schema is designed which has to be translated into a logical database schema following a translation algorithm. According to our proposal, the application of a *translation algorithm* implies the execution of a set

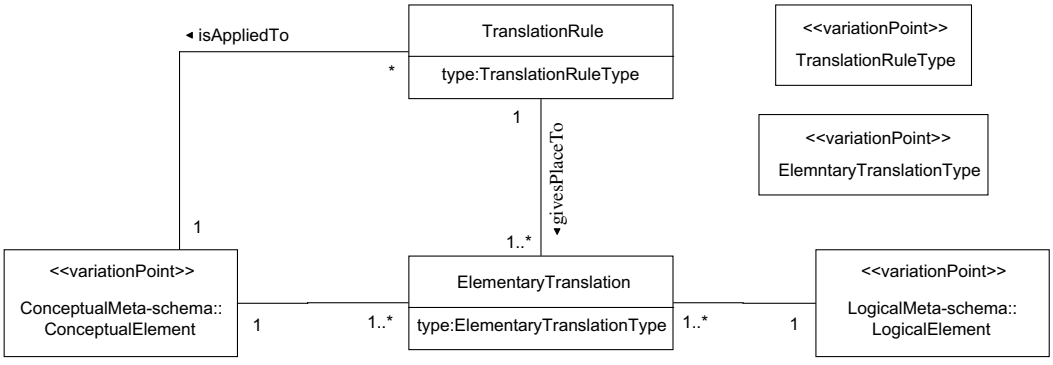

Fig. 3. Translation metamodel pattern.

of translation rules creating: (1) the elementary translations that relate the conceptual elements and the logical ones, (2) the logical elements of the logical schema and (3) the extensional schema.

In general, different translation rules can be applied for a conceptual element. In our approach, the translation rule to be applied can be chosen by the DBA or data designer. In this way, the translation process for each translated element consists of five steps:

Step (a): The DBA or data designer selects the translation rule to be applied to the instance of the initial conceptual schema that is going to be translated.
Step (b): The translation rule selected in step (a) is stored in the translation base.
Step (c): The translation rule is applied. Several conceptual elements can be involved in the translation process giving rise to the corresponding logical elements which are stored in the logical component.
Step (d): For each conceptual element translated into a logical element an elementary translation is added to the translation base.
Step (e): The correspondence rules are applied in order to determine the schema of the extensional component.

Steps (b) and (d) are not included in the traditional translation proposals, and they are the steps that guarantee the traceability of the process. Once the translation algorithm is applied, the database can be used and, in particular, the extensional schema can be populated.

For various reasons, the data structure may need to be changed. In this case, the data designer must issue the appropriate evolution transformations to the conceptual schema. When an evolution occurs the processors of the architecture come into play in order to keep the consistency among the different components.

Firstly, the conceptual processor reacts to external events received from the environment. Within our architecture we consider two types of external events, (1) those that perform a modification of the conceptual schema and (2) those that change the translation rule applied to an instance of the initial conceptual schema during the translation process. Each one of these types is handled in a different way.

With regard to (1) the external events performing a change in the conceptual schema, these are defined as a set of conceptual transformations. We propose to define this set taking into account the conceptual metamodel so that this set is a set of model transformations (adding, deleting or modifying elements of the conceptual schema). We propose to classify the transformations into primitive (which cannot be defined as a composition of the others) and compound transformations. In this respect, it is important to note that each transformation would imply not only schema modifications but also database extension changes.

In a context in which the database extension would not have to be considered, every transformation performing a schema modification would be a compound transformation since every modification could be defined as a composition of transformations deleting the schema elements to be modified and transformations creating the new elements. This is not suitable in a context such as ours, in which schema changes must be propagated to the extension, since the deletion of a schema element would imply the elimination of its instances, which could not be recovered.

Each event performing a conceptual schema modification is handled by the conceptual processor which checks its validity according to the restrictions imposed by the conceptual metaschema. If it is valid, the transformation is executed by the conceptual processor performing the modification of the conceptual database schema according to the semantics of the transformation.

With regard to (2) the external events changing the applied translation rule, these are also handled by the conceptual processor but the change does not modify the conceptual schema so that they are directly issued to the translation processor.

The modifications requested by the DBA or data designer by means of external events must be propagated to the rest of the components so that their stored information must be appropriately changed. As for the translation and logical components, the approach is the same as that in the conceptual one. A set of translation transformations and a set of logical transformations are defined taking into account the corresponding metamodels.

The conceptual changes are propagated to the translation and logical components by means of *propagation rules*. Each one of these rules determines the translation and logical transformations that must be performed

by the corresponding processors. Afterwards, the logical changes are propagated to the extensional component by means of the *correspondence rules*.

The approach followed in order to specify the propagation and the correspondence rules is the same. Both have to determine the changes to be performed in some elements as a result of a previous event. We have considered that this pattern of action can be easily specified by means of an Event–Condition–Action (ECA) rule [13] adapting appropriately the meaning of each one of its components.

Propagation rules. As for the propagation rules, the meaning of their components is the following:

1. The *event* that triggers the rule: this is a conceptual transformation.
2. The *condition* that determines whether the rule action should be executed: this is expressed in terms of the conceptual elements involved in the transformation and in terms of the applied translation rules.
3. The *action*: the translation and logical transformations that must be performed are determined.

The propagation rules must be defined in such a way that, for each conceptual transformation, the propagation rules to be executed are determined univocally. When a conceptual change is issued, the propagation rules whose event matches with the conceptual transformation and for which the condition of the rule is true are searched. In the action part of the ECA rules, the changes in the translation base and the logical schema are made.

By means of the execution of a propagation rule, the consistency between the conceptual and logical components is kept and the translation base is updated in order to reflect the new trace of the translation process after the evolution. In this way, the translation base now reflects the correspondence between the new conceptual and logical schemas according to the translation rules selected by the DBA or data designer. The conditions of the propagation rules (and the conditions of the correspondence rules which are explained later) are carefully designed in order to ensure that the changes performed do not interfere with each other.

Correspondence rules. Regarding the correspondence rules, the meaning of their components is the following:

1. The *event* that triggers the rule: this is a logical transformation.
2. The *condition* that determines whether the rule action should be executed: this is expressed in terms of the logical elements involved in the transformation.
3. The *action*: this is the set of procedures which generate SQL sentences that change the extensional schema and the data in a consistent way.

Like the propagation rules, the correspondence rules must be defined in such a way that, for each logical transformation, the correspondence rules to be executed are determined univocally. In the action part of the ECA rules, the changes in the schema of the database and in its extension are made. In particular, the extension modifications are performed by *data load procedures*. Each data load procedure has a precondition associated to it which states the circumstances in which it can be applied. In this respect, the main issue is that whenever a data load is used in an ECA rule, the condition of the ECA rule must ensure that the precondition of the data load holds.

It must be noted that this is the only case in which the processor changes not only the information base but also the schema of the component. But this is not surprising since this component is at one level of abstraction lower than the others so that the schema is a model whereas in the other cases they are metamodels (which are not changed in our proposed architecture).

Definition of evolution architecture instances. The evolution architecture we have defined is generic in the sense that its definition is independent of the particular languages that will be used in each one of the levels of abstraction involved (conceptual, logical and physical). This feature has been achieved by including a metamodel level which allows the architecture to be independent of specific modeling languages. The artifacts to be used in each component of the architecture must be determined in order to apply the architecture to a specific situation (that is to say, with specific modeling languages). We will use the term *evolution architecture instance* when the different generic elements of the architecture are settled giving rise to a particular evolution framework.

In order to determine an evolution architecture instance nine artifacts must be established:

1. The *conceptual metamodel* of the chosen conceptual modeling language.
2. The *translation metamodel* which follows the proposed translation metamodel pattern.
3. The *logical metamodel* of the chosen logical modeling language.
4. The *translation algorithm* which translates a conceptual model to a logical one.
5. The set of *conceptual transformations* for evolving the conceptual schema.
6. The set of *translation transformations* for evolving the translation base.
7. The set of *logical transformations* for evolving the logical schema.
8. The set of *propagation rules* from the conceptual schema to the logical schema, through the translation base.
9. The set of *correspondence rules* between the logical schema and the database state.

Note that the conceptual and logical transformations establish not only the syntax of the transformation (the structural mapping) but also its semantics (the instance mapping). For this reason, in the same way as is stated in [15], each conceptual or logical transformation is defined as a pair of mappings. The first mapping determines the way in which a (conceptual or logical, respectively) source schema is transformed into another schema, and the second mapping determines the way in which any instance of a source schema is transformed into an instance of the resultant schema.

In the next section, an example of an evolution architecture instance is given, so that all the nine artifacts are established.

4. Case study: EER and relational models

As we have said previously, the architecture has been developed with the property of being independent of any particular modeling technique, by means of the adoption of a metamodeling approach. However, in order to clarify the way the architecture works we will be considering a particular example, and for this reason some specific techniques must be established. For this example, we have chosen the most common techniques in the context of the database field: an enhanced entity-relationship (EER) model as the conceptual modeling technique and the relational model as the logical modeling technique. Another example of an application of MeDEA can be found in [8], where UML Class Diagram is used as conceptual modeling technique and XML Schema is used as logical modeling technique.

In this section, we present the particular artifacts that shape the evolution architecture instance with regard to the EER and relational models. Previously we introduce an example to which we will apply this architecture instance and that allows us to explain the several artifacts more easily.

4.1. Running example

In order to illustrate an example of database evolution, we will use the schema of Fig. 4a as the EER schema, which has been obtained combining different examples included in [19]. In this example, entity types,

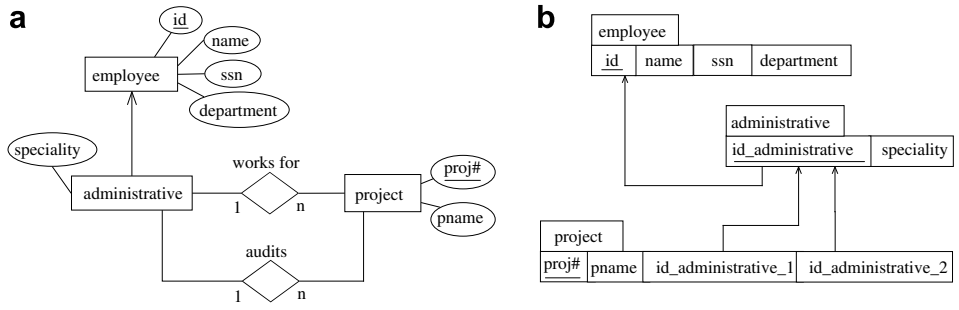

Fig. 4. Examples of conceptual and logical database schemas.

relationship types and attributes are represented, respectively, by rectangles, diamonds and ovals. This schema represents a company where it is perceived that there are employees and projects. Furthermore, the employees can be administrative staff (working for projects or auditing projects).

The proposed example of EER schema has been translated into a relational model (see Fig. 4b). Since there are different suitable translation rules, the DBA or data designer may choose the translation rules which are applied to each element type of the EER schema. For example, we suppose that the DBA has chosen to translate all the $1-n$ relationships types into foreign keys (represented by means of arrows) and the ISA hierarchies into a relation schema for each entity type.

4.2. Conceptual, translation and logical metamodels

For this case study, we have developed the different necessary metamodels following the guidelines of the Noesis Metamodeling Technique [11]. This technique provides a full detailed, structured definition of what a metamodel must be, based on a central notion of 'concept'. In particular, the Noesis technique indicates that every metamodel must include a graphical representation of the main concepts (namely a *support*), and provides a specific graphical notation. However, in this paper, for the sake of understandability, we use a UML notation for the diagrammatic part of the metamodels. Furthermore, we have included only the essential components of each metamodel.

The conceptual metamodel representing the EER model is shown in Fig. 5. It is worth noting several aspects of this metamodel. The metamodel includes a concept called *IsaSet*, which is used to refer to a set of ISA relationships that follows the same criterion of specialization and that shares the same 'parent'. This last property is considered a 'local constraint' in the Noesis Technique, and its expression using OCL as constraint language can be seen in Table 1. A similar notion to *IsaSet* has been included in the recent UML 2.0 specification [35]. Furthermore, the metamodel includes the concept of *IsaHierarchy*, which refers to a set of ISA relationships, all of them descending directly or indirectly from the same 'parent' (root). This definition is enforced by the constraints in Table 1.

The translation metamodel is shown in Fig. 6. This metamodel follows the pattern of translation metamodel we have described in Section 3.1, fixing the variation points to the particular case we are presenting as an example. In particular, the conceptual and logical elements have been fixed as EER and enhanced relational elements, respectively. With regard to the translation rules and elementary translations types, these will be explained in the next subsection where the translation algorithm is described.

Lastly, the logical metamodel representing the enhanced relational model is shown in Fig. 7. This metamodel conceptualizes the different elements that conform to a relational model [13]. Furthermore, it includes the notion of check constraint that is not included in the formal definition of relational model but that is

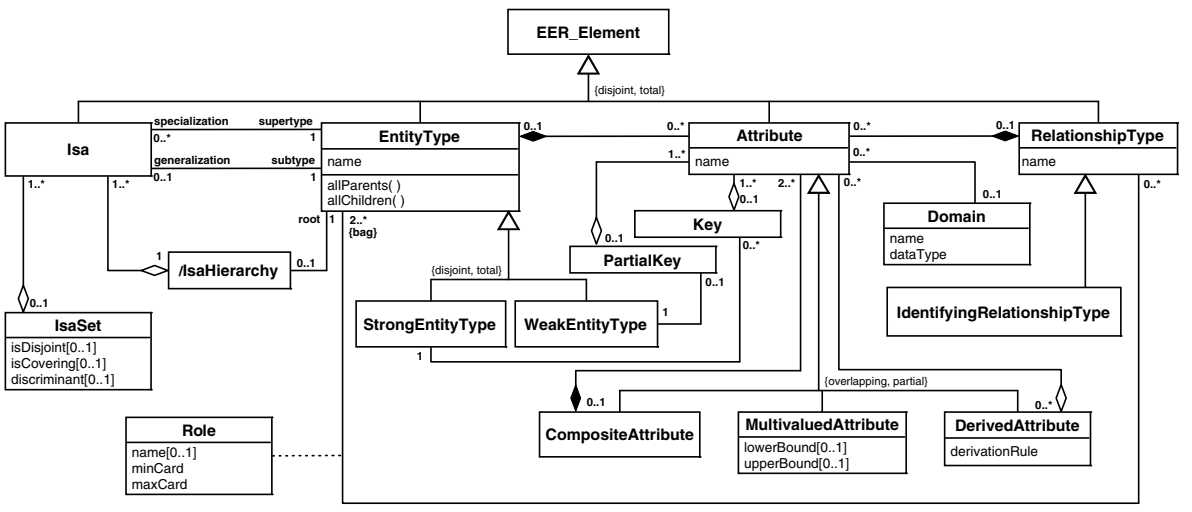

Fig. 5. Graphical representation of the support of our EER metamodel.

Table 1
Some local constraints of our EER metamodel

An IsaSet has only a supertype
`context IsaSet`
`inv:isa.supertype → asSet( ) → size( ) = 1`

The operation `allChildren( )` obtains the set of descendants of an EntityType
`context EntityType::allChildren( ):Set(EntityType)`
`body: self.speci.subtype → union (self.speci.subtype → collect(s|s.allChildren( )))`

The set of Isa of an IsaHierarchy is formed by the specializations of its root and the specializations of all the descendants of the root
`context IsaHierarchy::isa`
`derive: root.speci → union(root.allChildren( ).speci)`

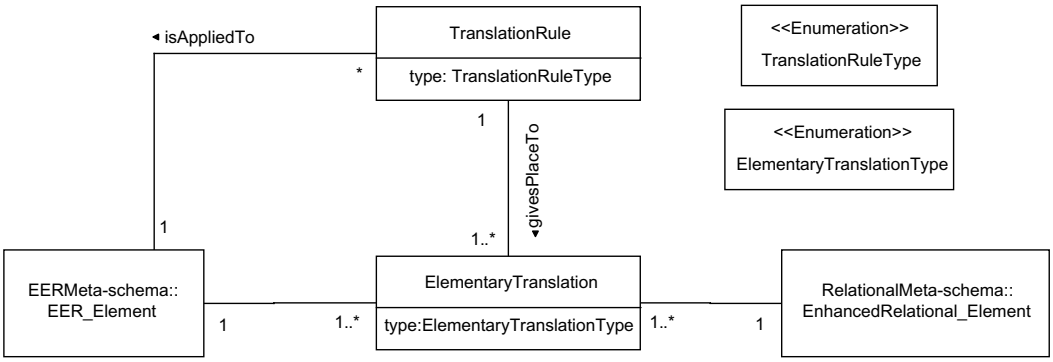

Fig. 6. Graphical representation of the support of our translation metamodel.

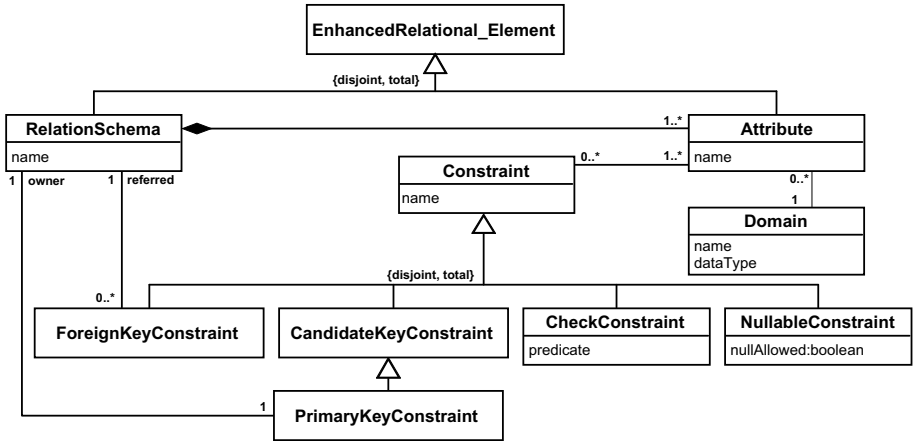

Fig. 7. Graphical representation of the support of our relational metamodel.

implemented in many database management systems. The reason for including this concept is that several conceptual elements are translated into a check constraint, for example, the attributes of covering and disjoint of an ISA set. Without this concept in the logical level, these constraints would be lost and could not be implemented in the extensional component.

4.3. Translation algorithm

In Fig. 8, the translation algorithm from EER to enhanced relational is shown. This algorithm has been determined following the translation algorithm proposed in [13]. As architect designers, we have decided

```
procedure EER2relational(EERSchema S)
begin
    for each entity type E not within isa hierarchy in S do
        TR = selectTranslationRule(E, 'entity type')
        applyEntityTypeTranslationRule(E, TR)
        Pk = primaryKey(E)
        TR = selectTranslationRule(Pk, 'primary key')
        applyConstraintTranslationRule(Pk, 'primary key', TR)
    end for
    for each isaHierarchy H in S do
        TR = selectTranslationRule(H, 'isaHierarchy')
        applyIsaHierarchyTranslationRule(H, TR)
    end for
    for each relationship type R in S do
        TR = selectTranslationRule(R, 'relationship type')
        applyRelationshipTypeTranslationRule(R, TR)
    end for
end
```

Fig. 8. EER to relational translation algorithm.

Table 2
Types of translation rules

Type	is the translation of...
EntityType01	an entity type into a relation schema. The relation schema has the same name as the entity type
EntityType02	an entity type into a relation schema. It has the same name as the entity type plus the prefix 'log'
RelSchemaPerEntityType01	an ISA hierarchy creating a relation schema per each entity type. Each relation schema has the same name as the entity type
RelSchemaForRoot01	an ISA hierarchy creating only a relation schema for the root. The relation schema has the same name as the root
ReifyRelationshipType01	a relationship type into a relation schema. The relation schema has the same name as the relationship type
NotReifyRelationshipType01	a relationship type into a foreign key

the elements to be translated as well as the translation rules that can be applied to each kind of element. Examples of translation rules are shown in Table 2.

For example, in Fig. 9 the algorithm of the RelSchemaPerEntityType01 translation rule for ISA hierarchies is shown. This rule creates a relation schema for each entity type of the hierarchy. Among the different actions executed by this rule we want to highlight the addition of elementary translations in order to store a trace of the translation. They appear explicitly as AddElementaryTranslation or they are included in the application of the translation rule for an EER element. As we have mentioned previously, the elementary translations can be of different types. The most representative types of elementary translations are shown in Table 3.

For example, we can apply the translation algorithm EER2Relational to the EER schema of Fig. 4a. Let us suppose that the following rules are chosen during the execution of the algorithm: EntityType01 for entity types, NotReifyRelationshipType01 for relationship types and RelSchemaPerEntityType01 for ISA hierarchies. Then the algorithm generates a set of elementary translations (regarding the example the most relevant of them are shown in Table 4) as well as the relational model of Fig. 4b.

4.4. Conceptual, translation and logical transformations

In order to carry out the EER schema evolution we have defined a family of conceptual transformations, taking into account other transformations proposed in the literature [1,32] (see Table 5). The full relation of conceptual transformations appears in the Appendix. A formal proof that ensures the completeness and soundness of this relation remains an open issue. We have developed similar tasks in a metamodeling context [11], and other works in the database evolution context can be used as Ref. [24].

Additionally, we have defined the transformation changeTranslationRule(E: conceptualElement, T_1: translationRule) which changes the translation rule applied to an element during the translation process.

```
translationRule RelationSchemaPerEntityType01(IsaHierarchy H)
begin
    R = getRootOfIsaHierarchy(H)
    TR = selectTranslationRule(R, 'entity type')
    applyEntityTypeTranslationRule(R, TR)
    RootPk = primaryKey(R)
    TR = selectTranslationRule(RootPk, 'primary key')
    applyConstraintTranslationRule(Pk, 'primary key', TR)
    for each isa I in H do
        E = subtype(I)
        TR = selectTranslationRule(E, 'entity type')
        applyEntityTypeTranslationRule(E, TR)
        T=table(E)
        R=createConstraint('pk', T)
        for each attribute A in RootPk do
            C = CreateAttribute(T)
            AddElementaryTranslation(A, C, 'PrimaryKey2AttributeSubtype')
            AddAttributeToConstraint(R, C)
        end for
        TR = selectTranslationRule(R,'primary key')
        applyConstraintTranslationRule(R, 'primary key', TR)
    end for
    for each isaSet S in H do
        TR = selectTranslationRule(S, 'isaSet')
        applyConstraintTranslationRule(S, 'isaSet', TR)
    end for
end
```

Fig. 9. Example of ISA translation rule.

Table 3
Types of elementary translations

Type	is the correspondence between...
EntityType2RelationSchema	an entity type and a relation schema
Attribute2Attribute	an entity type attribute and an attribute of a relation schema attribute
PrimaryKey2AttributeSubtype	an entity type primary key attribute and an attribute of a subtype relation schema
PrimaryKey2PrimaryKey	a conceptual primary key and a logical primary key
PrimaryKey2PrimaryKeySubtype	a conceptual primary key and a primary key of a subtype relation schema
RelType2RelSchema	a relationship type and a relation schema
Role2Attribute	a role of a relationship type and an attribute
Isa2ForeignKey	an ISA and a foreign key

Table 4
Some generated elementary translations

id	elemTransType	EERElement	RelElement	TransRuleApplied/To
16	Role2Attribute	auditsRoleAdmin	id_admin_2	NotReifyRelType01/audits
17	Role2Attribute	auditsRoleProject	proj#	NotReifyRelType01/audits
18	RelType2RelSch	audits	project	NotReifyRelType01/audits

The propagation of a conceptual transformation gives rise to a set of translation and logical changes expressed by means of the translation and logical transformations shown in Tables 6 and 7, respectively. Some examples of these transformations will be shown in the following sections.

4.5. Propagation rules

As we have explained in Section 3.2, by means of the propagation rules, each conceptual transformation is translated into a set of translation and logical changes expressed by means of the translation transformations

Table 5
Excerpt of family of conceptual transformations

PRIMITIVE CONCEPTUAL TRANSFORMATIONS
createIsa(*P*: entityType, *C*: entityType)
createIsaSet(*LI*: listOfIsas, [*Disc*: discriminant], [*Disj*?: boolean], [*Cov*?: boolean])
deleteRelationshipType(*R*: relationshipType)
deleteIsa(*I*: isa)
deleteIsaSet(*IS*: isaSet)
includeIsaInIsaSet(*I*: isa, *IS*: isaSet)
removeIsaFromIsaSet(*I*: isa, *IS*: isaSet)
COMPOUND CONCEPTUAL TRANSFORMATIONS
changeKey(*E*: entityType, *LA*: listOfAttributes)
= deleteKey(E) + addKey(E, LA)
createSubtypeAndIsaSet(*P*: entityType, *N*: nameOfEntityType, *Strong*?: boolean, [*LN*: listOfNamesOfAttributes], *LI*: listOfIsas, [*Disc*: discriminant], [*Disj*?: boolean], [*Cov*?: boolean])
= (createEntityType($N, Strong?$) as C) + forAll(N in LN do
(createAttribute(N) as A_i) + addAttributeToEntityType(A_i, C))
+ (createIsa(P, C) as *CisaP*) + createIsaSet($LI \cup \{CisaP\}, Disc, Disj?, Cov?$)

Table 6
Excerpt of family of translation transformations

PRIMITIVE TRANSLATION TRANSFORMATIONS
createTranslationRule(*E*: EERElement, *T*: translationRuleType)
deleteTranslationRule(*TR*: translationRule)
createElementaryTranslation(*E*: EERElement, *R*: relationalElement, *T*: elementaryTranslationType)
deleteElementaryTranslation(*ET*: elementaryTranslation)

Table 7
Excerpt of family of logical transformations

PRIMITIVE LOGICAL TRANSFORMATIONS
addAttributes(*LN*: listOfNamesOfAttributes, *LD*: listOfDomains, *E*: relationSchema)
addAndCopyAttributes(*E*1: relationSchema, *LN*:listOfNamesOfAttributes, *LD*:listOfDomains, *E*2: relationSchema, *LA*:listOfAttributes, *FK*: foreignKey)
deleteAttributes(*C*: listOfAttributes)
createRelationSchema(*N*: nameOfRelationSchema, *LN*1: listOfNamesOfAttributes, *LD*: listOfDomains, *LN*2: listOfNamesOfAttributesOfPK)
deleteRelationSchema(*E*: relationSchema)
createConstraint(*N*: nameOfConstraint, *CT*: constraintType, *S*: relationalElement, *T*: listOfRelationalElements)
deleteConstraint(*C*: constraint)

and the logical transformations. The modifications performed by each set of logical transformations re-establish the consistency of the logical schema with respect to the conceptual schema resulting after the application of each conceptual transformation. The set of logical transformations and their arguments are determined making use of the information stored in the translation component. Furthermore, the translation component is also updated in order to ensure the traceability of the translation process between the new conceptual and logical schemas.

For instance, let us consider three examples of evolution of the conceptual schema of Fig. 4a. One is the case in which the audits relationship type, together with its instances, must be deleted. As a second example, we are going to suppose that the attribute ssn is settled as a new key of the entity type employee. Finally, as another example of modification that affects the ISA hierarchy, a new entity type technician is added, being defined as a subtype of employee.

We also want to consider, as another type of change, the modification of the translation rules applied during the translation process. As an example of this change, let us consider that the user decides to change the way in which the ISA hierarchy of Fig. 4a is translated so that only a relation schema for the root of the hierarchy is established.

In order to perform these conceptual evolutions, the following conceptual transformations must be issued:

- deleteRelationshipType(audits),
- changeKey(employee, {ssn}),
- createSubtypeAndIsaSet(employee, 'technician', true, {'techgrade'}, {administrativeISAEmployee}, 'job', true, false),
- changeTranslationRule(employeeISAHierarchy, RelSchemaForRoot01).

Let us consider how the propagation algorithm deals with each one of these conceptual transformations.

Conceptual transformation deleteRelationshipType. The propagation of the first example of conceptual transformation seems to be very simple but it is more complex than it appears. Moreover, it will serve to illustrate the necessity of the translation component we include in our proposed architecture. According to the translation rule applied to this element, it is known that the features of the audits relationship type have been translated into different relational elements: a foreign key referencing the administrative table, the attribute of the foreign key and the table project to which the attribute belongs. The problem is that this relation schema contains two attributes(id_administrative_1 and id_administrative_2) verifying these conditions. If there is no information about the specific process followed for obtaining the logical schema of Fig. 4b, it is not known which attribute should be deleted. However, according to the elementary translations of Table 4, the audits relationship type has been translated into the relation schema project (elementary translation 18) by adding to it the attribute id_administrative_2 (elementary translation 16). So this is the attribute which must be deleted.

The propagation rule triggered by the deleteRelationshipType transformation appears in Table 8. Let us see how this propagation rule works for our running example. Line (1) of the action finds the foreign key of the relationship type audits and line (2) finds the attributes into which the relationship type has been translated. In line (3) the elementary translations reflecting the translation of the relationship type audits are deleted from the translation component. In line (4), the translation rules applied to the translation of the relationship type are deleted. Finally, in lines (5)–(7) the foreign key from project.id_administrative_2 to administrative.id_administrative is deleted and in line (8) the project.id_administrative_2 attribute is deleted.

Conceptual transformation changeKey. In this transformation, the attribute employee.ssn has to be settled as the new primary key of entity type employee. The propagation of this simple conceptual transformation is different depending on the translation rule selected by the user. This information is stored in the translation component, so that it is known that the translation of the primary key has been performed inside the translation rule RelSchemaPerEntityType01. This information together with the elementary translations are used by the propagation algorithm in order to determine the set of changes that must be performed in the logical schema.

When we change the key of the root of an ISA hierarchy, then, at the logical level, new attributes (which will be the new attributes of the primary key of the relation schema) must be added to each relation schema of each subtype of the ISA hierarchy.

Table 8
An example of propagation rule

Name	deleteForeignKeys
Event	deleteRelationshipType(R)
Condition	R has not been reified in the translation process
Action	(1) LFK ← getForeignKeysOfRelationshipType(R) (2) LC ← getAttributesOfRelationshipType(R) (3) deleteElementarytranslations(R) (4) deleteTranslationRules(R) (5) **for each** foreignKey FK **in** LFK **do** (6) deleteConstraint(FK) (7) **end for** (8) deleteAttributes(LC)

Conceptual transformation createSubtypeAndIsaSet. This transformation creates a new entity type technician, and is defined as a subtype of employee. Besides, it creates an isaSet with the subtypes technician and administrative and with the property isDisjoint set to true and the property isCovering set to false. As in the previous example, the propagation of this conceptual transformation takes into account that the ISA hierarchy has been translated by means of the translation rule RelSchemaPerEntityType01.

These two examples show that, thanks to our architecture, the logical changes are analyzed in terms of conceptual changes. As a consequence, for example, ISA evolution issues can be analyzed within the relational model (which lacks the ISA concept). From our point of view, this is one of the most noteworthy contributions of our approach.

Transformation changeTranslationRule. As a final evolution example, the translation rule to translate the ISA hierarchy is changed. The translation rule RelSchemaForRoot01 (only one relation schema for the root of the hierarchy) must be used instead of RelSchemaPerEntityType01 (one relation schema for each entity type).

4.6. Correspondence rules

The changes performed in the logical schema are propagated to the extensional component by means of the correspondence rules. As we have mentioned previously, the correspondence rules have to determine two types of SQL sentences: (1) those that modify the structure of the physical database and (2) those that change the extension of the database by means of *data load procedures*. Each data load procedure has a precondition associated to it so that whenever a data load is used in a correspondence rule the precondition must hold.

For example, the logical transformation addAndCopyAttributes must be translated into SQL sentences that, at the physical level, create the corresponding columns and that populate them with the values of other columns. This is the goal of the populateNewColumns correspondence rule (see Table 9) whose action executes (1) the procedure SQLNewColumns which determines the SQL sentences that add new columns to a table of the physical database and (2) the data load procedure columnDataLoad which determines the SQL sentences that populate the new columns. In this action, the function getColumnsForeignKey returns the columns of a foreign key and the function getReferredColumnsForeignKey returns the columns to which the columns of a foreign key refer.

The columnDataLoad procedure adds values to columns of a target table, these values are taken from columns of another source table (see Table 10). In this case the precondition establishes that a unique row exists in the source table each time values are obtained from the source columns. The verification of this precondition is ensured by the populateNewColumns correspondence rule thanks to the foreign key that exists from the target relation schema to the source relation schema.

For instance, in our running example, when the key of the root of the ISA hierarchy is changed by means of the conceptual transformation changeKey(employee, {ssn}) then, among other things, the following logical transformation is carried out in the logical schema:

addAndCopyAttributes(administrative, {'administrative.ssn'}, {integer}, employee, {employee.ssn}, AdmToEmplFK)

Table 9
An example of correspondence rule

Name	populateNewColumns
Event	addAndCopyAttributes(targetRelSchema, newTargetAttributes, datatypes, sourceRelSchema, sourceAttributes, targetToSourceFK)
Condition	targetToSourceFK is a foreign key from targetRelSchema to sourceRelSchema
Action	(1) SQLNewColumns(newTargetAttributes, datatypes, targetRelSchema) (2) columnDataLoad(getTargetColumns(newTargetAttributes), getSourceColumns(sourceAttributes), getTargetTable(targetRelSchema), getSourceTable(sourceRelSchema), getColumnsForeignKey(targetToSourceFK), getReferredColumnsForeignKey(targetToSourceFK))

Table 10
An example of data load procedure

Name	columnDataLoad
Parameters	targetColumns, sourceColumns, targetTable, sourceTable, targetJoinColumns, sourceJoinColumns
Precondition	In sourceTable exists an unique row that satisfies the join condition expressed in the Description section
Description	This procedure loads data in the (initially empty) targetColumns. For each row r of targetTable, the value of targetColumns is s.sourceColumns where s is the unique row of sourceTable satisfying the join condition s.sourceJoinColumns = r.targetJoinColumns
Code	`UPDATE targetTable SET targetColumns = (SELECT sourceColumns FROM sourceTable WHERE sourceJoinColumns = targetJoinColumns)`

In turn, this logical transformation triggers the correspondence rule populateNewColumns which executes the following procedures:

SQLNewColumns({administrative.ssn}, {INTEGER}, administrative)
columnDataLoad({administrative.ssn}, {employee.ssn}, administrative, employee, {administrative.id_administrative}, {employee.id})

The first action adds the column ssn to the table administrative. The second action loads the new column with data coming from the column employee.ssn. The particular context of this change ensures the precondition of the columnDataLoad procedure since the foreign key from administrative to employee allows a unique row satisfying the join condition to be determined.

For the complete running example, the generated SQL sentences can be seen in Table 11. Sentences (a)–(b) correspond to the logical changes due to the conceptual transformation deleteRelationshipType, sentences (c) to (n) to the transformation changeKey, sentences (o) to (r) to the transformation createSubtypeAndIsaSet and sentences (s) to the end to the transformation changeTranslationRule.

4.7. Implementation

In this section, we describe a specific implementation of our architecture. This implementation is based on the RDBMS Oracle 10g Release 2 and the programming language PL/SQL 10g Release 2 [34].

The processor of the conceptual component has been implemented as a set of PL/SQL procedures, one for each of the established conceptual transformations. For example, the procedure deleteRelationshipType accomplishes the task of deleting a relationship type at the conceptual level.

The processor of the translation component implements the propagation rules as a collection of PL/SQL triggers which are fired by the insert, delete or update operations performed on the conceptual database. For example, the propagation rule deleteForeignKeys of Table 8 has been implemented as a PL/SQL trigger named trgDeleteForeignKeys.

The processor of the logical component is also implemented as a collection of PL/SQL procedures and PL/SQL triggers which are fired by the insert, delete or update operations performed on the translation base. For example, as a consequence of the execution of the trigger trgDeleteForeignKeys the procedure deleteConstraint deletes the foreign key into which the audits relationship type was translated.

The extensional component stores the Oracle 10g database schema and its data. The processor of this component implements the correspondence rules as a collection of PL/SQL triggers which are fired by the insert, delete or update operations performed in the logical database. These triggers automatically generate and execute the SQL sentences against the Oracle 10g database in order to reach a consistent database state. For example, the correspondence rule populateNewColumns of Table 9 is implemented by means of the trigger trgPopulateNewColumns and is fired, for example, as a consequence of executing the procedure changeKey(employee, {ssn}), which changes the primary key of the entity type employee.

The implementation has been tested in several prototypes developed within our master studies. From this experience, we have drawn several conclusions about the helpfulness of the architecture. The first conclusion is that the architecture has proven to be of great value for keeping an updated conceptual schema, unlike traditional systems in which the conceptual–logical correspondence is rapidly lost. Secondly, it has been very

Table 11
Generated SQL sentences which change the physical component

a. ALTER TABLE project DROP CONSTRAINT ProjToAdmFK2
b. ALTER TABLE project DROP COLUMN id_administrative_2
c. ALTER TABLE employee DROP CONSTRAINT EmplPK CASCADE
d. ALTER TABLE employee ADD CONSTRAINT EmplPK PRIMARY KEY (ssn)
e. ALTER TABLE administrative ADD ssn INTEGER
f. UPDATE administrative SET administrative.ssn = (SELECT employee.ssn FROM employee WHERE employee.id = administrative.id_administrative)
g. ALTER TABLE administrative DROP CONSTRAINT AdmPK CASCADE
h. ALTER TABLE administrative ADD CONSTRAINT AdmPK PRIMARY KEY(ssn)
i. ALTER TABLE administrative ADD CONSTRAINT AdmToEmplFK FOREIGN KEY (ssn) REFERENCES employee(ssn)
j. ALTER TABLE project ADD ssn INTEGER
k. UPDATE project SET project.ssn = (SELECT administrative.ssn FROM administrative WHERE administrative.id_administrative = project.id_administrative_1)
l. ALTER TABLE project DROP COLUMN id_administrative_1
m. ALTER TABLE project ADD CONSTRAINT projToAdmFK1 FOREIGN KEY (ssn) REFERENCES administrative(ssn)
n. ALTER TABLE administrative DROP COLUMN id_administrative
o. CREATE TABLE technician(ssn INTEGER, techgrade VARCHAR2(10))
p. INSERT INTO technician VALUES (10,'Superior')
q. ALTER TABLE technician ADD CONSTRAINT TechPK PRIMARY KEY (ssn)
r. ALTER TABLE technician ADD CONSTRAINT TechToEmplFK FOREIGN KEY (ssn) REFERENCES employee(ssn)
s. ALTER TABLE employee ADD job VARCHAR2(10)
t. ALTER TABLE employee ADD techgrade VARCHAR2(10)
u. UPDATE employee e SET e.techgrade = (SELECT t.techgrade FROM technician t WHERE t.ssn = e.ssn), e.job = 'tech' WHERE EXISTS (SELECT * FROM technician t WHERE t.ssn = e.ssn)
v. ALTER TABLE employee ADD speciality VARCHAR2(50)
w. UPDATE employee e SET e.speciality = (SELECT s.speciality FROM administrative s WHERE s.ssn = e.ssn), e.job = 'secr' WHERE EXISTS (SELECT * FROM administrative s WHERE s.ssn = e.ssn)
x. ALTER TABLE employee ADD CONSTRAINT ckeck2 CHECK(((job = 'tech') AND (techgrade IS NOT NULL)) OR ((job = 'secr') AND (speciality IS NOT NULL)) OR ((job IS NULL) AND (techgrade IS NULL) AND (speciality IS NULL)))
y. ALTER TABLE project DROP CONSTRAINT projToAdmFK1
z. ALTER TABLE project ADD CONSTRAINT projToEmplFK1 FOREIGN KEY (ssn) REFERENCES employee(ssn)
aa. DROP TABLE administrative
ab. DROP TABLE technician

helpful in terms of reliability in respect of changes made in the physical database in the face of the error–prone manual restructuring. However, we have also found that it is not quite so helpful as expected when we had to apply complex transformations that we have not been able to express as a combination of more primitive ones. As a consequence, some manual adjustments of the components storing the schemas have been made when some of these primitives should have been issued against the architecture.

5. Related work

Database evolution has been widely discussed in the literature and therefore very varied approaches have been proposed. The evolution of object-oriented databases and relational databases, including the propagation of changes automatically down to the extension of the database, has received great attention and the research results have been included in prototypes or in commercial DBMS (see, for example [1]). However, they lack the consideration of a conceptual level which allows the designer to work at a higher level of abstraction [20].

In [16], an abstract framework which takes into account both conceptual and logical levels is presented and the necessity of automatically propagating down (forward strategy) the changes performed at the conceptual level is stated. The various papers dealing with forward engineering mainly differ in two aspects (which are related) (1) the mechanisms they use for preserving the consistency between levels and (2) the way they address the propagation of the conceptual changes down to the logical schema and to the extension.

With regard to the consistency between levels, several mechanisms have been proposed in the literature for keeping data in a consistent state after schema changes occur. The data conversion mechanism we have

adopted is the strict conversion method, in which changes are immediately propagated to the data. As is pointed out in [31], this mechanism, in contrast to the other two options (lazy and logical), has longer schema modification time but reduces subsequent data access time. In general three dimensions can be considered in order to evaluate our architecture from a performance point of view: database size, database modification time and database access time. We estimate that MeDEA is particularly well suited for small or medium size databases and for large databases that are not expected to have many changes. For these kinds of databases the database modification time is compensated for by a faster database access time. However, in large databases with very frequent changes, the data modification time is so long that we would not recommend the use of our architecture.

As for the way the conceptual changes are propagated down to the logical schema and to the extension, several approaches have been proposed. For example, a taxonomical approach is followed in [30], which proposes a taxonomy of changes for ER structures and the impact of these changes on relational schemes is analyzed. However, this paper does not study how to reflect the schema evolution in the extension of the database. We tackle this problem by ensuring the traceability of the translation process between levels, so that information stored about the translation process allows the propagation to be done more easily.

The necessity of capturing information about translations performed between models has been recognized in the literature and very varied approaches have been proposed. In the Model-Driven Architecture (MDA) approach [21], mapping techniques between Platform Independent Models (PIM) or between Platform Specific Models (PSM) are considered as association classes so that a mapping will be represented as an object (relating models) and not only as a link. With respect to PIM to PSM mappings, the MDA approach proposes 'incremental consistency' as a desirable feature [18]. Within our architecture this property is achieved by means of the translation component.

The innovative trend of model management [2,5] also advocates the representation of mappings by means of objects. It is argued that this reification is often needed for satisfactory expressiveness [2]. This proposal aims to be generic in the sense that it can be applied to different kinds of models while our work is specific to database evolution issues.

In the Query-View-Transformation (QVT) proposal [27] the traceability elements are deduced from the translation algorithm (and so they depend on it). However, in our proposal the traceability component is previously given, independently of the algorithm. The algorithm can be changed and the component continues to be the same.

In [15], the traceability is achieved by storing all the sequence (called history) of operations performed during the translation of the conceptual schema into a logical one. In this way the mappings affected by the changes can be detected and modified, whereas the rest can be re-executed without any modification. The main difference between this approach and ours is the type of information stored for assuring traceability. Whereas in [15] the idea is to store the history of the process performed (probably with redundancies), in our case the goal of the elementary translations is to reflect the correspondence between the conceptual elements and the logical ones, so there is no room for redundancies. A synthetic version is also proposed in [15], more similar to ours since they propose to store the link between concepts of different levels. The difference is that the transformations carried out are not stored so that there is room for ambiguities, the DBA or data designer intervention being necessary in order to perform the evolution process. Another difference is that we follow a metamodeling approach.

A metamodeling approach is also proposed in [20,26,23]. In the case of [20] only a conceptual metamodel is considered whereas we also make use of a logical and a translation metamodel. With respect to [26], the authors make use of a metamodeling approach with a different goal since the paper deals with the definition of a query language for evolving information systems. In [23], a generalization of the traditional information system notion similar to ours has been proposed. However, some differences with respect to our proposal are worth noting. Firstly, in [23] not only data modeling is taken into account (as in our case) but also process and behavior specification. Secondly, in [23] only the conceptual level is under consideration so that the proposed architecture includes only one information system. Finally, the information processor of an information system is concerned with the modification of the structure and also of the population, instead of using different information processors for each one of these processes as we propose.

6. Conclusions and future work

In this paper we have presented MeDEA, a generic architecture for database evolution. We have shown how the metamodeling perspective that has been used in the development of this architecture allows it to be used whatever the chosen modeling techniques. Moreover, and with the aim of presenting the functionality of the architecture more completely, in this paper, we have presented a specific application setting of MeDEA. In this detailed case study the most usual database modeling languages are used, that is the Entity-Relationship Model as conceptual modeling language and the relational model as logical modeling language.

One of the most noteworthy characteristics of MeDEA is the inclusion of a *translation component*, devoted to storing, by means of a model-based structure, the translation of conceptual schemas to logical ones. When an evolution event is issued against a conceptual schema, the translation component allows that changes to be propagated towards the logical schema, the physical schema and the extension, in such a way that the only elements that are modified are precisely those elements that are affected by the evolution task. This approach has several advantages, from an improvement of the performance of the evolution process to a saving in the number of possible errors that could be made if elements not strictly involved were included in the process.

In this work, we assume that all the artifacts involved (conceptual schema, logical schema, extension) are available when an evolution task is carried out, in such a way that the problem is addressed from a 'forward maintenance' point of view. One of the most obvious lines of future work is the analysis of the problem of 'backward maintenance' which is closely related to inverse engineering issues. We will have to investigate whether MeDEA fits adequately to this setting or not, and in the latter case what kind of adaptations should be made to the architecture.

Another research line is the generalization to other contexts. In its conception, MeDEA is an architecture tied to the database field. However, similar evolution problems arise when the target artifacts are different kinds of storage structures other than databases, such as for instance XML documents. In particular, a potentially fruitful area would be the study of evolution problems in healthcare contexts. Some previous work undertaken by our research group [6] has proven that evolution issues naturally occur within this field.

Acknowledgements

The authors of the paper greatly appreciate the comments of the referees, since their comments have been of great help in improving the paper.

Appendix A. Conceptual transformations

A.1. Conceptual transformations of creation

1. *createEntityType*(*N*: name, *Strong*?: boolean)
2. *createRelationshipTypeAndRoles*(*N*: name, *LE*: listOfEntityTypes, [*LN*: listOfNamesOfRoles], *LC*: listOfPairsOfCardinalities, *Ident*?: boolean)
3. *createAttribute*(*N*: name, [*D*: domain], [*DT*: data type])
4. *addAttributeToEntityType*(*A*: attribute, *E*: entityType)
5. *addAttributeToRelationshipType*(*A*: attribute, *R*: relationshipType)
6. *specifyCompoundAttribute*(*A*: attribute, *LA*: listOfAttributes)
7. *specifyMultivaluedAttribute*(*A*: attribute, [*LB*: lowerBound], [*UB*: upperBound])
8. *specifyDerivedAttribute*(*A*: attribute, *DR*: derivationRule, [*LA*: listOfAttributes])
9. *createKeyOfStrongEntityType*(*E*: entityType, *LA*: listOfAttributes)
10. *createPartialKeyOfEntityType*(*E*: entityType, *LA*: listOfAttributes)
11. *createISa*(*P*: entityType, *C*: entityType)
12. *createIsaSet*(*LI*: listOfIsas, [*Disc*: discriminant], [*Disj*?:boolean], [*Cov*?:boolean])

A.2. Conceptual transformations of elimination

13. *deleteEntityType*(*E*: entityType)
14. *deleteRelationshipType*(*R*: relationshipType)
15. *deleteAttribute*(*A*: attribute)
16. *deleteAttributeFromEntityType*(*A*: attribute, *E*: entityType)
17. *deleteAttributeFromRelationshipType*(*A*: attribute, *R*: relationshipType)
18. *deleteKeyFromStrongEntityType*(*E*: entityType, *K*: key)
19. *deletePartialKeyFromEntityType*(*E*: entityType, *PK*: partialKey)
20. *deleteISa*(*I*: isa)
21. *deleteIsaSet*(*IS*: isaSet)

A.3. Conceptual transformations of modification

22. *renameEntityType*(*E*: entityType, *N*: name)
23. *renameRelationshipType*(*R*: relationshipType, *N*: name)
24. *renameAttribute*(*A*: attribute, *N*: name)
25. *renameDomain*(*D*: domain, *N*: name)
26. *renameRole*(*R*: role, *N*: name)
27. *includeAttributeInCompositeAttribute*(*A*: attribute, *CA*: compositeAttribute)
28. *includeAttributeInDerivedAttribute*(*A*: attribute, *DA*: derivedAttribute)
29. *includeAttributeInKey*(*A*: attribute, *K*: key)
30. *includeAttributeInPartialKey*(*A*: attribute, *PK*: partialKey)
31. *includeIsaInIsaSet*(*I*: isa, *IS*: isaSet)
32. *removeAttributeFromCompositeAttribute*(*A*: attribute, *CA*: compositeAttribute)
33. *removeAttributeFromDerivedAttribute*(*A*: attribute, *DA*: derivedAttribute)
34. *removeAttributeFromKey*(*A*: attribute, *K*: key)
35. *removeAttributeFromPartialKey*(*A*: attribute, *PK*: partialKey)
36. *removeIsaFromIsaSet*(*I*: isa, *IS*: isaSet)
37. *changeCardinalityOfRole*(*R*: role, *C*: pairOfCardinalities)
38. *changeDataTypeOfDomain*(*D*: domain, *DT*: dataType)
39. *changeLowerBoundOfMultivaluedAttribute*(*A*: attribute, *LB*: lowerBound)
40. *changeUpperBoundOfMultivaluedAttribute*(*A*: attribute, *UB*: upperBound)
41. *changeDerivationRuleOfDerivedAttribute*(*A*: attribute, *DR*: derivationRule)
42. *changeIsDisjointOfIsaSet*(*IS*: isaSet, *Disj*?: boolean)
43. *changeIsCoveringOfIsaSet*(*IS*: isaSet, *Cov*?: boolean)
44. *changeDiscriminantOfIsaSet*(*IS*: isaSet, *Dis*: discriminant)
45. *changeDomainOfAttribute*(*A*: attribute, *D*: domain, [*DT*: dataType])
46. *turnAttrributeIntoEntityType*(*A*: attribute, *E*: entityType, *Strong*?: boolean)
47. *turnRelationshipTypeIntoEntityType*(*R*: relationshipType, *E*: entityType, *Strong*?: boolean)
48. *inheritanceOfAttribute*(*I*: isa, *A*: attribute)
49. *inheritanceOfRelationshipType*(*I*: isa, *R*: relationshipType)
50. *disinheritanceOfAttribute*(*I*: isa, *A*: attribute)
51. *disinheritanceOfRelationshipType*(*I*: isa, *IS*: isaSet)
52. *includeEntityTypeInRelationshipType*(*E*: entityType, *R*: relationshipType, [*N*: nameOfRole], *C*: pairOfCardinalities)
53. *removeEntityTypeFromRelationshipType*(*E*: entityType, *R*: relationshipType)

A.4. Compound conceptual transformations

54. *changeKey*(*E*: entityType, *LA*: listOfAttributes) = deleteKey(E) + addKey(E, LA)

55. *createSubtypeAndIsaSet*(*P*: entityType, *N*: nameOfEntityType, *Strong*?: boolean, [*LN*: listOfNamesOfAttributes], *LI*: listOfIsas, [*Disc*: discriminant], [*Disj*?: boolean], [*Cov*?: boolean]) = (createEntityType(N, *Strong*?) as C) + forAll(N in LN do (create Attribute(N) as A_i) + addAttribute ToEntityType(A_i, C)) + (createIsa(P, C) as *CisaP*) + createIsaSet($LI \cup \{CisaP\}$, *Disc*, *Disj*?, *Cov*?)

References

[1] L. Al-Jadir, M. Léonard, Multiobjects to ease schema evolution in an OODBMS, in: T.W. Ling, S. Ram, M.L. Lee (Eds.), Conceptual Modeling, ER-98, LNCS, vol. 1507, Springer, 1998, pp. 316–333.

[2] P.A. Bernstein, Applying model management to classical meta data problems, in: First Biennial Conference on Innovative Data Systems Research – CIDR 2003, Online Proceedings, 2003.

[3] M. Clauss, Generic modeling using UML extensions for variability, in: Proceedings of the Workshop on Domain-specific Visual Languages, OOPSLA 2001, 2001, pp. 11–18.

[4] K.T. Claypool, E.A. Rundensteiner, G.T. Heineman, ROVER: a framework for the evolution of relationships, in: A.H.F. Laender, S.W. Liddle, V.C. Storey (Eds.), Conceptual modeling, ER-2000, LNCS, vol. 1920, Springer, 2000, pp. 409–422.

[5] K.T. Claypool, E.A. Rundensteiner, Gangam: a transformation modeling framework, in: International Conference on Database Systems for Advanced Applications – DASFAA 2003, IEEE Computer Society, 2003, pp. 47–54.

[6] E. Domínguez, J. Lloret, B. Pérez, A. Rodríguez, A.L. Rubio, M.A. Zapata, Model-driven development based transformation of stereotyped class diagrams to XML schemas in a healthcare context, in: Proceedings of the First International Workshop on Conceptual Modelling for Life Sciences Applications (CMLSA 2007), LNCS, vol. 4802, Springer, 2007, pp. 44–53.

[7] E. Domínguez, J. Lloret, A.L. Rubio, M.A. Zapata, Elementary translations: the seesaws for achieving traceability between database schemata, in: Proceedings of the Workshop on Evolution and Change in Data Management (ECDM 2004), LNCS, vol. 3289, Springer, 2004, pp. 377–389.

[8] E. Domínguez, J. Lloret, A.L. Rubio, M.A. Zapata, Evolving XML schemas and documents using UML class diagrams, in: Proceedings of the Database and Expert Systems Applications (DEXA 2005), LNCS, vol. 3558, Springer, 2005, pp. 343–352.

[9] E. Domínguez, J. Lloret, A.L. Rubio, M.A. Zapata, Evolving the implementation of ISA relationships, in: EER Schemas, Proceedings of the Workshop on Evolution and Change in Data Management (ECDM 2006), LNCS, vol. 4231, Springer, 2006, pp. 237–246.

[10] E. Domínguez, J. Lloret, M.A. Zapata, An architecture for managing database evolution, in: Proceedings of the Workshop on Evolution and Change in Data Management (ECDM 2002), LNCS, vol. 2784, Springer, 2002, pp. 63–74.

[11] E. Domínguez, M.A. Zapata, Towards a situational method engineering technique, Information Systems 32 (2) (2007) 181–222.

[12] E. Domínguez, M.A. Zapata, J.J. Rubio, A conceptual approach to meta-modelling, in: A. Olivé, J.A. Pastor (Eds.), Advanced Information Systems Engineering, CAISE'97, LNCS, vol. 1250, Springer, 1997, pp. 319–332.

[13] R.A. Elmasri, S.B. Navathe, Fundamentals of Database Systems, fourth ed., Addison-Wesley, 2003.

[14] J.J. van Griethuysen (Ed.), Concepts and Terminology for the Conceptual Schema and the Information Base, Publ. ISO/TC97/SV5-N695, Mars 1982.

[15] J.M. Hick, J.L. Hainaut, Database application evolution: a transformational approach, Data and Knowledge Engineering 59 (3) (2006) 534–558.

[16] J.L. Hainaut, V. Englebert, J. Henrard, J.M. Hick, D. Roland, Database evolution: the DB-MAIN approach, in: P. Loucopoulos (Ed.), Entity-Relationship Approach – ER'94, LNCS, vol. 881, Springer-Verlag, 1994, pp. 112–131.

[17] T.A. Halpin, H.A. Proper, Database schema transformation and optimization, in: M.P. Papazoglou (Ed.), Object-Oriented Entity-Relationship Modelling Conference – ER'95, LNCS, vol. 1021, Springer-Verlag, 1995, pp. 191–203.

[18] A. Kleppe, J. Warmer, W. Bast, MDA Explained. The Model Driven Architecture: Practice and Promise, Addison-Wesley, 2003.

[19] A.H.F. Laender, M.A. Casanova, A.P. de Carvalho, L.F.G.G.M. Ridolfi, An analysis of SQL integrity constraints from an entity-relationship perspective, Information Systems 10 (4) (1994) 331–358.

[20] J.R. López, A. Olivé, A framework for the evolution of temporal conceptual schemas of information systems, in: B. Wangler, L. Bergman (Eds.), Advanced Information Systems Engineering, CAiSE 2000, LNCS, vol. 1789, 2000, pp. 369–386.

[21] J. Miller, J. Mukerji (Eds.), MDA Guide Version 1.0, Object Management Group, Document number omg/2003-05-01, May 1, 2003.

[22] C. Nicolle, D. Benslimane, K. Yetongnon, Multi-data models translations in interoperable information systems, in: J. Mylopoulos, Y. Vassiliou (Eds.), Advanced Information Systems Engineering, CAISE'96, LNCS, vol. 1080, Springer, 1996, pp. 1–21.

[23] J.L.H. Oei, H.A. Proper, E.D. Falkenberg, Evolving information systems: meeting the ever-changing environment, Information Systems Journal 4 (3) (1994) 213–233.

[24] R.J. Peters, M.T. Özsu, An axiomatic model of dynamic schema evolution in objectbase systems, ACM Transactions on Database Systems 22 (1) (1997) 75–114.

[25] H.A. Proper, Data schema design as a schema evolution process, Data and Knowledge Engineering 22 (2) (1997) 159–189.

[26] H.A. Proper, Th.P. van der Weide, Information disclosure in evolving information systems: taking a shot at a moving target, Data and Knowledge Engineering 15 (1995) 135–168.

[27] OMG, MOF 2.0 Query/Views/Transformations RFP, ad/2002-0410, 2002. Available from: <http://www.omg.org>.

[28] E. Rahm, P.A. Bernstein, An online bibliography on schema evolution, ACM SIGMOD Record 35 (4) (2006) 30–31.

[29] B. Ramesh, Factors influencing requirements traceability practice, Communications of the ACM 41 (12) (1998) 37–44, December.
[30] J.F. Roddick, N.G. Craske, T.J. Richards, A taxonomy for schema versioning based on the relational and entity relationship models, in: R.A. Elmasri, V. Kouramajian, B. Thalheim (Eds.), Proceedings of the 12th International Conference on the Entity-Relationship Approach, LNCS, vol. 823, Elsevier, 1994, pp. 137–148.
[31] J.F. Roddick, A survey of schema versioning issues for database systems, Information Software Technology 37 (7) (1995) 383–393.
[32] E. Rodríguez, A. Abelló, M. Oliva, F. Saltor, C. Delgado, E. Garvi, J. Samos, On operations along the generalization/specialization dimension, in: International Workshop on Engineering Federated Information Systems, EFIS, IOS Press, 2001, pp. 70–83.
[33] A.S. da Silva, A.H.F. Laender, M.A. Casanova, An approach to maintaining optimized relational representations of entity-relationship schemas, in: B. Thalheim (Ed.), Conceptual Modeling – ER'96, LNCS, vol. 1157, Springer-Verlag, 1996, pp. 292–308.
[34] Oracle Corporation, Oracle Database, Application Server, and Collaboration Suite Documentation. <http://tahiti.oracle.com>. Last visited July 2007.
[35] OMG, UML 2.0 Superstructure Spec., formal/05-07-04, 2005. <www.uml.org>.
[36] W.M.N. Wan-Kadir, P. Loucopoulos, Relating evolving business rules to software design, Journal of Systems Architecture 50 (7) (2004) 367–382.

Eladio Dominguez is a Full Professor of Computer Science at the University of Zaragoza, Spain, and academician of the Royal Academy of Sciences of Zaragoza. He was previously Associate Professor of Topology at University of Zaragoza and the Polytechnical University of Madrid. He has taught mathematics at the Universities of Sevilla and Valencia. Dr. Dominguez received his Ph.D. in 1974 from the University of Zaragoza. His main research interests are in Digital Topology, Information Systems and Phenomenology.

Jorge Lloret is a Lecturer in Computer Science at the University of Zaragoza, Spain. He received his B.Sc. in Mathematics in 1991 and his Ph.D. in computer science in 1997 from the University of Zaragoza. His main research interest is focused on database evolution.

Angel Luis Rubio is a Lecturer in Computer Science at the University of La Rioja, Spain. He received his B.Sc. degree in Mathematics from the University of Zaragoza, Spain, in 1994, and his Ph.D. in computer science from the University of La Rioja in 2002. His current research interests are focused on metamodeling and models evolution (specially database evolution).

Maria Antonia Zapata is a Lecturer in Computer Science at the University of Zaragoza, Spain. She received her B.Sc. in Mathematics in 1988 and her Ph.D. in Computer Science in 1994 from the University of Zaragoza. Her main research interests are in model-driven development, database evolution and clinical decision support systems.