

Model–Driven, View–Based Evolution of Relational Databases^{*}

Eladio Domínguez¹, Jorge Lloret¹, Ángel L. Rubio²,
and María A. Zapata¹

¹ Dpto. de Informática e Ingeniería de Sistemas.
Facultad de Ciencias. Edificio de Matemáticas.
Universidad de Zaragoza. 50009 Zaragoza. Spain
`{noesis,jlloret,mazapata}@unizar.es`

² Dpto. de Matemáticas y Computación. Edificio Vives.
Universidad de La Rioja. 26004 Logroño. Spain
`arubio@dmc.unirioja.es`

Abstract. Among other issues, database evolution includes the necessity of propagating the changes inside and between abstraction levels. There exist several mechanisms in order to carry out propagations from one level to another, that are distinguished on the basis of when and how the changes are performed. The strict mechanism, which implies the immediate realization of modifications, is a time-consuming process. In this paper we propose a solution that is closer to the lazy and logical mechanisms, in which changes are delayed or not finally realized, respectively. This solution makes use of the notion of view. The use of views allows the data not to be changed if it is not necessary and facilitates carrying out changes when required.

1 Introduction

There are two main interrelated issues that are very frequently highlighted in existing database evolution literature: information consistency and propagation of modifications. For example, both issues are described in [10] and included within a more general activity called *managing core schema evolution*. Information consistency is concerned with the crucial point of ensuring the lossless of any semantic information when an evolution task is carried out. Propagation of modifications must be considered along two dimensions with regard to abstraction layers. On the one hand, within a *horizontal* dimension the changes in some part of a schema can (and usually must) be conveyed to other parts of the schema. On the other hand, within a *vertical* dimension the changes in a schema situated in an abstraction layer must be propagated to the corresponding schema in other abstraction layers (for instance, from the relational level downwards to the extensional level).

^{*} 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 Aragon and by the European Social Fund.

In [2] we have presented a proposal in which information consistency is preserved by using a database evolution architecture, called MeDEA, framed in a forward engineering context. This architecture has been developed from a model-driven perspective, since it has been recognized that schema evolution is one of the applications of model management [1]. With respect to propagation of modifications, different authors classify several ways of carrying out propagations. For instance, Roddick in [8] distinguishes three kinds of *data conversion mechanisms*: strict, lazy and logical. In the *strict* mechanism changes in the schema are propagated to the data immediately. In the *lazy* mechanism, routines that can perform the changes in the data are assembled, but they are used only when required. Lastly, a system of screens is created to translate the data to the appropriate format at access time (without converting data) in the *logical* mechanism. Another classification is proposed in [7] where screening, conversion and filtering techniques are described. Although this second classification is tied to object-based systems, a rather close parallel can be drawn between the *screening* technique and the lazy data conversion mechanism, the *conversion* technique and the strict mechanism, and the *filtering* technique and the logical mechanism. Our above-mentioned architecture, as presented in [2], uses the strict data conversion mechanism.

In any case, there exists a ‘common line’ in the literature that states that the best solution for the propagation of modifications implies making a balance between avoiding the modification of data until it is indispensable and not increasing excessively the complexity of the database. As a step to obtain such a solution, in this paper we propose a new application of our architecture in which views are used. The use of views allows the modification of data not to be initially realized, and therefore, our approach fits into the logical procedure. A noteworthy contribution of this idea is that when, for some reason, the modifications have to be translated into the physical database, this task is facilitated since views *codify* the changes undergone at the logical level. In this sense, our adopted solution is closer to the lazy mechanism and the screening technique.

The structure of the paper is as follows. Section 2 first reviews the basic ideas of our architecture for database evolution, and then presents the adaptation of the architecture to the view-based approach. In Section 3 we discuss the effect of the conceptual changes when a view-based approach is followed. We devote Section 4 to detailing the technical aspects of the view based propagation algorithm. Finally, related work, some conclusions and future work appear in Sections 5 and 6.

2 View-Based Database Evolution

2.1 Brief Description of MeDEA

MeDEA is a MEtamodel-based Database Evolution Architecture we have presented in [2]. In this section, we describe briefly the meaning and purpose of the components of MeDEA (see [2] for details).

MeDEA has four components (see Figure 1): *conceptual component*, *translation component*, *logical component* and *extensional component*. The *conceptual component* captures machine-independent knowledge of the real world. In this work, it deals with EER schemas. The *logical component* captures tool-independent knowledge describing the data structures in an abstract way. In this paper, it deals with schemas from the relational model by means of standard SQL. The *extensional component* captures tool dependent knowledge using the implementation language. Here, this component deals with the specific database in question, populated with data, and expressed in the SQL of Oracle. One of the main contributions of our architecture is the *translation component*, that not only captures the existence of a transformation from elements of the conceptual component to others of the logical one, but also stores explicit information about the way in which specific conceptual elements are translated into logical ones.

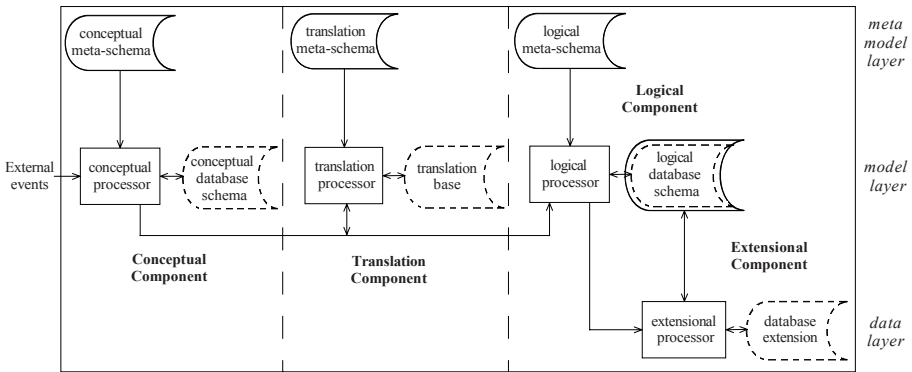


Fig. 1. MeDEA Database Evolution Architecture

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 [14]) 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 [14]). All the other elements are situated at the model layer [14].

In order to apply MeDEA to the specific situation of this paper (that is, to EER and relational schemas) nine artifacts must be established: three metamod- els (EER, translation and relational), a translation algorithm from EER schemas to relational schemas, three sets of evolution transformations (one set for each metamodel), one set of propagation rules and one set of correspondence rules.

Once these nine artifacts are given (see Section 2.2) the way of working of the architecture is as follows: given an EER schema in the conceptual component, the translation algorithm is applied to it and creates: (1) the logical database schema, (2) a set of elementary translations in the translation component and (3) the physical database schema.

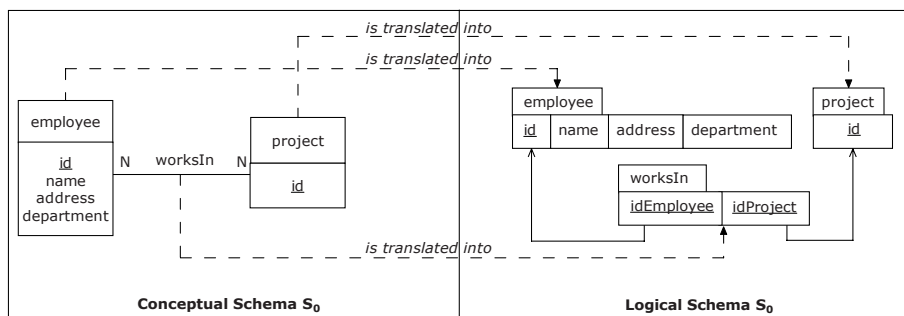


Fig. 2. Conceptual and logical schemas

For instance, as running example, we will consider an initial EER schema S_0 (see Figure 2) with employees and projects. Each employee has an **id**, **name**, **address** and **department** where (s)he works. Each project has an **id** and many employees can work in many projects. The application of the translation algorithm to this EER schema gives rise to: (1) the relational schema with three relation schemas shown in Figure 2, (2) elementary translations storing which EER elements are translated into relational elements (some of the generated elementary translations are represented as dashed lines in Figure 2) and (3) the SQL sentences of Oracle that create the relational schema.

Then, the physical database is populated with data. 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 EER schema. The propagation algorithm propagates the changes to the other components. To be precise, the EER schema changes are propagated to the translation and relational components by means of the propagation rules. Afterwards, the relational schema changes are propagated to the Oracle database by means of the correspondence rules.

In Sections 3 and 4, an evolution example, using the EER schema S_0 as starting point, is shown. Four evolution transformations are issued which are propagated to the relational schema, the physical schema and the data.

2.2 Application of MeDEA Using a View-Based Evolution Approach

In [2] we applied MeDEA to EER and relational schemas following a strict data conversion mechanism. The main contribution of this paper is the application of this architecture using a lazy and logical data conversion mechanism, so that the changes are delayed as far as possible. In order to reach this goal, views are used for evolving relational databases whenever the conceptual changes allow this to be done, avoiding the overhead of strict data conversion. However, if the number of views is increased too much, query processing performance can be downgraded.

This change in the way of managing the evolution processes has led us to change some of the artifacts proposed in [2]. To be precise, the EER meta-model, the translation algorithm and the set of conceptual changes remain the same. It should be noted that, during the execution of the translation algorithm, the user may choose the translation rules which are applied to each element of the EER schema giving place to relational structures. For example, for relationship types the user can choose, among others, between the **relSchemaPerRelType** translation rule ('a relation schema for each relationship type') or **relSchemaPerNNRelType** translation rule ('a relation schema for the N-N relationship type and foreign key for other cardinalities of the relationship'). Additional details about the unchanged artifacts can be found in [2].

The rest of the artifacts proposed in [2] have been changed in order to allow us to perform a lazy and logical data conversion mechanism. In particular, the relational metamodel has been modified including in it the concept of **view** and distinguishing between **base** and **derived** attributes (see Figure 3). A view is described by its **name**, a **predicate** and a **checkOption**. Let us notice that a view can be based on relation schemas and views, and that its attributes can be base or derived attributes. These changes in the logical metamodel oblige us to include new logical transformations regarding the new concepts (see Table 1).

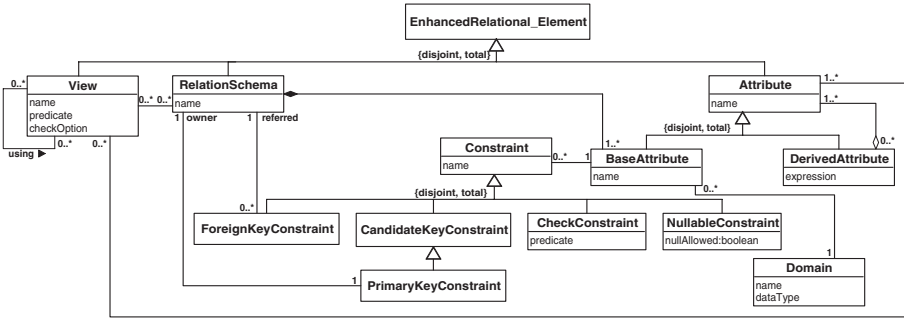


Fig. 3. Graphical representation of the relational Metamodel

The modification of the logical metamodel also implies the inclusion in the translation component of new types of elementary translations. In this way, the **EntityType2View** type reflects the translation of an entity type into a view and the **Attribute2ViewAttribute** type reflects the translation of a conceptual attribute into an attribute of a view. Moreover, the **RelType2View** type reflects the translation of a relationship type into a view. Here we have a constraint according to which for each entity or relationship type there is only one relation schema or view in the logical schema. As a consequence, there is only one elementary translation for each entity or relationship type.

The rest of the paper is devoted to studying the effects of the conceptual changes, to explain the view based propagation algorithm and the propagation rules, bringing to light the advantages of this new proposal.

Table 1. New logical transformations

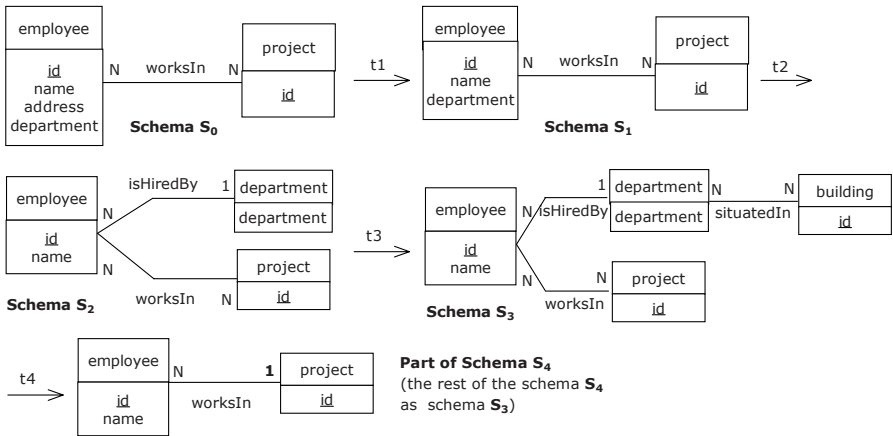
createView(*N*: name; [*LR*: listOfRelationSchemas]; [*LV*: listOfViews]; *LA*: listOfAttributes; *P*: predicate; *CO*: checkOption)
createDerivedAttribute(*N*: name; *LA*: listOfAttributes; *E*:expression)
deleteView(*V*: view)
deleteDerivedAttribute(*DA*: derivedAttribute)
renameView(*V*: view; *N*: name)
renameDerivedAttribute(*DA*: derivedAttribute; *N*: name)
includeRelationSchemaInView(*V*: view; *R*: relationSchema)
includeViewInView(*V*: view; *W*: view)
includeAttributeInView(*V*: view; *A*: attribute)
includeAttributeInDerivedAttribute(*DA*: derivedAttribute; *A*: attribute)
removeRelationSchemaInView(*V*: view; *R*: relationSchema)
removeViewInView(*V*: view; *W*: view)
removeAttributeInView(*V*: view; *A*: attribute)
removeAttributeInDerivedAttribute(*DA*: derivedAttribute; *A*: attribute)
changePredicate(*V*: view; *P*: predicate)
changeCheckOption(*V*: view; *CO*: checkOption)
changeExpression(*DA*: derivedAttribute; *E*: expression)

3 Study of the Effects of the Conceptual Changes

For strict database conversion, we have proposed in [2] a propagation algorithm. Its input is a set of conceptual transformations which change the conceptual schema. Its output is a set of SQL sentences which add, drop or modify tables, columns and constraints as well as a set of data load procedures. The SQL sentences and data load procedures are executed against the physical database so that the new physical database schema conforms with the new conceptual schema.

With the aim of applying the idea of delaying as far as possible the strict conversion of data, we have studied how the propagation algorithm should be modified (giving place to the *view based propagation algorithm*). For this purpose, we have considered the set of MeDEA conceptual transformations which can be applied to a conceptual schema and we have determined how to change the effect of these transformations in the components of the architecture when the concept of view is used. The modifications we have found have been incorporated into the view based propagation algorithm which we describe in the next section.

So as to illustrate our study, we will use the evolution example of Figure 4 in which, starting from the schema S_0 , the attribute **address** is deleted, the attribute **department** is transformed into an entity type **department**, a new entity type **building** and a new relationship type **situatedIn** between **department** and **building** are set. Finally, the cardinality of the relationship type **worksIn** is changed and an employee can work at most in one project.



```

t1: deleteAttribute(employee.address)
t2: turnAttributeIntoEntityType(employee.department,department, true, 'isHiredBy')
t3: createEntityType('building', true)
    createAttribute('building.id',INTEGER)
    addAttributeToEntityType(building.id,building)
    createKeyOfStrongEntityType(building,(id))
    createRelationshipTypeAndRoles('situatedIn',(department, building), null,((0,N),(0,N)), true)
t4: changeCardinalityOfRole(rEmployeeWorksIn, (0,1))

```

Fig. 4. Evolved schemas

Revision of the effects of the conceptual transformations. We briefly describe the effects of some of the conceptual changes and we only describe in depth the transformation for adding a relationship type and for changing the cardinality of a role.

Add an attribute to an entity type. If the entity type is transformed into a relation schema, add an attribute to the relation schema. If the entity type is transformed into a view, add the corresponding attribute to the original relation schema of the entity type as well as to the view.

Delete an attribute from an entity type. Mark as deleted the elementary translations of the attribute. Add a new view or modify an existing one in such a way that the corresponding attribute does not appear.

Add an entity type. Create a new relation schema for the entity type.

Drop an entity type. Marked as deleted the elementary translations of the elements of the entity type as well as those of the relationship types in which the entity type participates.

Add a relationship type. The transformation is `createRelationshipTypeAndRoles(n,le,ln,lc,ident?)` and it creates a new relationship type with name *n*, the list *le* of participants, the list *ln* of names of roles, the list *lc* of cardinalities and information about whether the relationship type is an identifying

Table 2. Representations of relationship types

Case	e	f	rt	Example
	has been transformed into...		is transformed into...	
1	r1	r2	r3, new relation schema or relation schema from one of the participants	transf. t_3
2	v1	r2	r3, new relation schema	transf. t_4
3	v1	r2	v3, view based on v1	
4	v1	r2	r3=r2	
5	v1	v2	r3, new relation schema	transf. t_2
6	v1	v2	v3, view based on v1 or v2	

relationship type. Let us suppose that the participants are **e** and **f**. The different representations of the relationship type are shown in Table 2 where **vx** stands for view and **rx** stands for relation schema. Each one of these cases is contemplated in at least one propagation rule.

The first case of Table 2 will be treated by a propagation rule of the initial propagation algorithm because both of the participants have been transformed into relation schemas and no views are involved. For the rest of the cases, new propagation rules are defined, which contemplate the fact that at least one of the participants is transformed into a view. These rules represent the relationship types as a relation schema (case 2, 4 and 5) or as a view (cases 3, 6).

The first situation (relationship type as a relation schema) happens when we need to augment the information capacity of the schema. For example, in transformation t_3 , we add a new relationship type **situatedIn** for having the information about the departments and buildings where they are located. As the relationship type is N–N, we need a new relation schema to store these relationships.

The second situation (relationship type as a view) appears when we can embed the relationship type into the base relation schema of some of the participants. For example, in transformation t_2 , the relationships between the employees and the projects can be seen through a view on relation schema **employee**. In this respect, we have found two cases:

1. The view for the relationship type is the same as the view for one of the participants, but adequately modified. For example, this happens in transformation t_4 where the view for the relationship type **worksIn** is the same as the view for the entity type **employee**.
2. The view for the relationship type is based on the view for one of the participants but does not modify this view. For example, this happens in transformation t_2 where the view for the entity type **department** is built on the view for the entity type **employee**.

Drop a relationship type. Mark as deleted the elementary translations of the elements of the relationship type.

Change the cardinality of a relationship type. If we change the cardinality of a relationship type from N-1 to N-N two situations can appear. If previously there is a relation schema for the relationship type, no change must be made other than modifying the elementary translation for the relationship type. The new logical element for the relationship type is the previously existing relation schema. Otherwise, the information capacity of the logical schema must be increased and a new relation schema must be created. Views can not be used in either of the two situations.

However, for changing the cardinality of a relationship type from N-N to N-1 when the data allow this to be done, we can create a view for the relationship type and add a unique constraint in the base relation schema of the relationship type to cope with the new cardinality constraint. Database operations continue to be done on the original relation schema of the N-N relationship type. An example can be seen in transformation t_4 .

Remarks. When a conceptual transformation drops a conceptual element, the corresponding elementary translations are marked as deleted but the relational structures are not dropped. The corresponding columns are no longer accessible from the conceptual level, and the columns or tables will be dropped when restructuring database tasks are performed.

The restructuring consists of changing the logical schema in such a way that only tables are left and the consistency between levels is kept. The key point here, and one of the most noteworthy contributions of this paper, is that the use of view converts this task into an easy one. The reason is that the views keep a memory of the changes undergone by the logical schema. Then, creating a new table for an entity or relationship type will basically consist of transforming the view into a table.

For example, the view `vEmployee`, after transformation t_4 , is gathering the information of the entity type `employee(attributes id, name)`, of the relationship type `isHiredBy` (by means of the attributes `id, department`) and of the relationship type `worksIn` (by means of the attributes `id, idProject`).

When this view has to be transformed into a table, two steps related with employees must be made at the physical level:

1. The `department` column values are replaced by identifiers for the departments
2. The view `vEmployee` is easily transformed into a table with a SQL sentence like `CREATE TABLE employee2 AS SELECT * FROM vEmployee`

The analysis we have just done is integrated into the existing propagation rules or into new ones, giving place to the view based propagation algorithm (see next section).

4 View Based Propagation Algorithm

In our new perspective of delaying the conversion of the data, the changes in the EER schema are propagated to the physical database by means of the view








elementary_translation				Legend for elementary translations
id	type	conceptual_element	logical_item	
 1	RelType2RelSchema	worksIn	worksIn	 marked as deleted
 2	Attribute2Attribute	employee.id	worksIn.idEmployee	 affected
 3	Attribute2Attribute	project.id	worksIn.idProject	
 4	RelType2View	worksIn	vEmployee	 added

Fig. 5. Some elementary translations involved in the `changeCardinalityOfRole` example

Table 3. Sketch of the subalgorithm for the logical level

```

INPUT: Set of changes in the translation component and information
about the conceptual changes
OUTPUT: Changes on the logical component
For each translation change c of the INPUT
  Find the logical propagation rule r of the
  view based logical propagation rules set (if there is one) such that:
    - the event of r matches the translation change c AND
    - the condition of r is met
  Execute the action of r
end For
If such a rule not exists, repeat the same with the rules of the
original set of propagation rules
End For

```

based propagation algorithm. This is split into subalgorithms for the translation, for the logical and for the extensional component.

Subalgorithm for the translation component. This receives as input a set of changes in the conceptual component. Its output is a set of added, marked as deleted and affected elementary translations.

The initial subalgorithm for the translation component has been modified in order to give place to ‘marked as deleted’ elementary translations instead of deleted elementary translations.

For instance, when the `changeCardinalityOfRole` transformation of Figure 4 is executed, the elementary translation 1(Figure 5) is marked as deleted, the elementary translations 2 and 3 are affected, and the elementary translation 4 is added.

Subalgorithm for the logical component. The changes in the translation component are the input for the propagation subalgorithm for the logical component (see the sketch in Table 3).

It is based on the set of existing logical propagation rules together with the new view based logical propagation rules. Both kinds of rules are ECA rules [4] and the latter have the following meaning for its components:

Table 4. An example of logical propagation rule

1. **Name:** modifyView
 2. **Event:** newElementaryTranslation(*e*)
 3. **Condition:** (the conceptual change is `changeCardinalityOfRole(r,(0,1))`)
AND (the type of *e* is `RelType2View`) AND (the conceptual element of *e* is the relationship type from which *r* is a role)
 4. **Action:**

```

et←getEntityType(r)
rt←getRelType(r)
source←getRelSchemaOrView(et)
target←getRelSchemaOrView(rt)
targetAttributes←getAttribs(rt,et)
sourceJoinAttributes←getEntityTypeID(et)
targetJoinAttributes←getRelTypeID(rt,et)
predicate←equality(sourceJoinAttributes,targetJoinAttributes)
includeRelationSchemaInView(source, target)
includeAttributeInView(source, targetAttributes)
changePredicate(source, predicate)

```
1. The event that triggers the rule. This is a transformation on the translation component.
 2. The condition that determines whether the rule action should be executed. This is expressed in terms of the conceptual and logical elements involved in the affected elementary translations and in terms of the conceptual change.
 3. The actions. These include procedures which create, delete or modify logical views.

An example of a view based logical propagation rule can be seen in Table 4.

The subalgorithm for the logical component has two sets of propagation rules. One of them is the set of logical propagation rules coming from the initial propagation algorithm. The second set is the set of newly created propagation rules with the purpose of delaying as far as possible the strict conversion of data.

For each elementary translation (whether it is added, marked as deleted or affected) the subalgorithm searches the only rule (if there is one) in the second set whose event is the same as the event of the elementary translation and whose condition evaluates to true. Then, the subalgorithm executes the actions of the rule changing the logical component. If there is no rule in the second set satisfying the above conditions, then a rule in the first set is searched for. If found and its condition evaluates to true, the actions of the rule are applied. If there is no rule in any of the sets, no rule is fired and the logical component is not changed for this elementary translation.

Our algorithm at work. When the conceptual transformation t_4 `changeCardinalityOfRole('rEmployeeWorksIn',(0,1))` is applied to the schema S_3 of Figure 4, the subalgorithm for the translation component, among other things, (1) adds a new elementary translation (number 4 in Figure 5) with type

Table 5. Logical changes after applying the conceptual transformations

- (1) **Logical changes for deleteAttribute(employee.address)**
 (a) `createView('vEmployee', employee, null, (id,name,department),null,null);`
- (2) **Logical changes for turnAttributeIntoEntityType(employee.department,department, true, 'isHiredBy')**
 (b) `createView('vDepartment',null,vEmployee,department,'department IS NOT NULL',null);`
- (3) **Logical changes for createEntityType('building', true), createAttribute('building.id',INTEGER), addAttributeToEntityType(building.id,building) and createKeyOfStrongEntityType(building,(id))**
 (c) `createRelationSchema('building','id',INTEGER,'id')`
- (4) **Logical changes for createRelationshipTypeAndRoles('situatedIn',(department, building), null,((0,N),(0,N)), true)**
 (d) `createRelationSchema('situatedIn',('department','idBuilding'), (VARCHAR(30),INTEGER),('department','idBuilding'))`
 (e) `createConstraint('fk3','fk',situatedIn.idBuilding,building.id)`
- (5) **Logical changes for changeCardinalityOfRole(rEmployeeWorksIn, (0,1))**
 (f) `includeRelationSchemaInView(vEmployee,worksIn)`
 (g) `includeAttributeInView(vEmployee,worksIn.idProject)`
 (h) `changePredicate(vEmployee,'employee.id=worksIn.idEmployee')`

Table 6. Generated SQL sentences which apply a view based evolution

- a. `CREATE OR REPLACE VIEW vEmployee AS SELECT id, name, department FROM employee`
- b. `CREATE OR REPLACE VIEW vDepartment AS SELECT DISTINCT department FROM vEmployee WHERE department IS NOT NULL`
- c. `CREATE TABLE building(id INTEGER, name VARCHAR2(30), CONSTRAINT pk4 PRIMARY KEY (id))`
- d. `CREATE TABLE situatedIn(department VARCHAR2(20), idBuilding INTEGER, CONSTRAINT pk5 PRIMARY KEY(department, idBuilding))`
- e. `ALTER TABLE situatedIn ADD CONSTRAINT fk3 FOREIGN KEY (idBuilding) REFERENCES building(id)`
- f. `ALTER TABLE worksIn ADD CONSTRAINT uniq1 UNIQUE(idEmployee)`
- g. `CREATE OR REPLACE VIEW vEmployee AS SELECT id, name, department, idProject FROM employee, worksIn WHERE employee.id=worksIn.idEmployee`

RelType2View to store the fact that the relationship type `worksIn` is now being converted into a view, (2) marks as deleted the elementary translation 1 in Figure 5 and (3) detects the affected elementary translations numbers 2 and 3. These elementary translations are the input for the subalgorithm for the logical component. For each one of them, the subalgorithm searches the logical rule (if any) in the second set with the same event as the event of the translation change and for which the condition of the rule evaluates to true.

For the newly added elementary translation number 4, the logical propagation rule `modifyView` is fired because its event is the same as the event of the added elementary translation and its condition evaluates to true. As a consequence, the view `vEmployee` is modified as can be seen in (f) to (h) of Table 5. With respect to the affected and marked as deleted elementary translations no changes are made at the logical level because no rule is fired.

Finally, the extensional propagation subalgorithm (not shown here) takes as input the changes in the logical component and changes the extensional component by means of SQL sentences. For the complete running example, the SQL sentences generated by this subalgorithm can be seen in Table 6. Sentence (a) corresponds to the conceptual transformation `deleteAttribute`, sentence (b) to transformation `turnAttributeIntoEntityType`, sentence (c) to transformations `createEntityType`, `createAttribute`, `addAttributeToEntityType` `createKeyOfStrongEntityType`, sentences (d) and (e) to transformation `createRelationshipTypeAndRoles` and sentences (f) and (g) to transformation `change CardinalityOfRole`.

5 Related Work

The development of techniques that enable schema changes to be accommodated as seamlessly and as painlessly as possible is an apparent theme in recent schema evolution research [9]. This is the main aim that guides the proposal we present in this paper. We started from a schema evolution proposal we presented in [2], which follows a strict data conversion mechanism. The problem of the strict (or eager, or early) conversion method is that it takes longer schema modification time [8] causing a heavy load to the system [12]. With the goal of avoiding this problem, in this paper we have modified our previous proposal by using a lazy and logical data conversion mechanism. The lazy (or deferred) and logical mechanism has the advantage that changes can be made more rapidly [8].

We propose to achieve the lazy and logical conversion mechanism by means of views, so that a logical modification of data is performed whenever the conceptual changes allow this to be done. Views have been proposed by several authors [10] as a way of performing different schema evolution processes. The most common use of views has been to simulate schema versioning [13]. View mechanism has also been employed to create personal schemas when a database is shared by many users [11]. However, these approaches lack the consideration of a conceptual level which allows the designer to work at a higher level of abstraction.

The novelty of our approach is that the schema evolution is performed at a conceptual level so that users interact with schema changes at a high level, such as is demanded in [9]. To our knowledge the approaches that propose a conceptual level (for example [5]) do not propose a lazy and logical conversion mechanism. In [6] views are used within a framework with conceptual level, but they are involved during the translation process instead of during the evolution process as in our case. For this reason, as far as we are aware, this is the first time that views are used for achieving a lazy and logical conversion mechanism within a framework where different levels of abstraction are involved following a metamodel perspective [3].

6 Conclusions and Future Work

One of the main problems that arise when database evolution tasks must be realized is that of propagation of modifications. In particular, the propagation in a

vertical sense (from conceptual to logical, from logical to extensional levels) can be carried out following different data conversion mechanisms. The strict mechanism, in which changes are propagated immediately, is the most frequently used. For instance, this is the method we used in [2] where we have presented MeDEA, a database evolution architecture with traceability properties. However, it has been recognized that the strict mechanism takes longer schema modification time and causes a heavy workload to the system. As a step to solve this problem, in the present paper we have proposed a new application of MeDEA, by making use of the notion of view. The main advantage of using views is that it is not necessary to use the strict mechanism. On the contrary, the changes in data do not have to be initially realized, which sets a scenario closer to the logical data conversion mechanism. Furthermore, views codify internally the changes, in such a way that these changes can be realized explicitly, if required, in a similar way to the lazy mechanism.

In the present paper we have used the MeDEA architecture by fixing EER and Relational as techniques for the conceptual and logical levels, respectively. This decision has allowed us to use views in a natural way. A possible future line of work is the study of other kind of techniques where views or similar concepts can be used to carry out evolution tasks.

References

1. Bernstein, P.A.: Applying Model Management to Classical Meta Data Problems. In: First Biennial Conference on Innovative Data Systems Research- CIDR 2003, Online Proceedings (2003)
2. Domínguez, E., Lloret, J., Rubio, A.L., Zapata, M.A.: MeDEA: A database evolution architecture with traceability. *Data & Knowledge Engineering* 65, 419–441 (2008)
3. Domínguez, E., Zapata, M.A.: Noesis: Towards a Situational Method Engineering Technique. *Information Systems* 32(2), 181–222 (2007)
4. Elmasri, R.A., Navathe, S.B.: *Fundamentals of Database Systems*, 4th edn. Addison-Wesley, Reading (2003)
5. Hick, J.M., Hainaut, J.L.: Database application evolution: A transformational approach. *Data and Knowledge Engineering* 59(3), 534–558 (2006)
6. Mork, P., Bernstein, P.A., Melnik, S.: Teaching a Schema Translator to Produce O/R Views. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) *ER 2007*. LNCS, vol. 4801, pp. 102–119. Springer, Heidelberg (2007)
7. Peters, R.J., Tamer Özsu, M.: An Axiomatic Model of Dynamic Schema Evolution in Objectbase Systems. *ACM Trans. on Database Systems* 22(1), 75–114 (1997)
8. Roddick, J.F.: A survey of schema versioning issues for database systems. *Information Software Technology* 37(7), 383–393 (1995)
9. Roddick, J.F., de Vries, D.: Reduce, Reuse, Recycle: Practical Approaches to Schema Integration, Evolution and Versioning. In: Roddick, J.F., Benjamins, V.R., Si-said Cherfi, S., Chiang, R., Claramunt, C., Elmasri, R.A., Grandi, F., Han, H., Hepp, M., Lytras, M., Mišić, V.B., Poels, G., Song, I.-Y., Trujillo, J., Vangenot, C. (eds.) *ER Workshops 2006*. LNCS, vol. 4231, pp. 209–216. Springer, Heidelberg (2006)

10. Ram, S., Shankaranarayanan, G.: Research issues in database schema evolution: the road not taken, working paper # 2003-15 (2003)
11. Ra, Y.G., Rundensteiner, E.A.: A Transparent Schema-Evolution system based on object-oriented view technology. *IEEE Transactions of Knowledge and Data Engineering* 9(4), 600–624 (1997)
12. Tan, L., Katayama, T.: Meta Operations for Type Management in Object-Oriented Databases – A Lazy Mechanism for Schema Evolution. In: Kim, W., et al. (eds.) *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pp. 241–258. North Holland, Amsterdam (1990)
13. Tresch, M., Scholl, M.H.: Schema Transformation without Database Reorganization. *ACM SIGMOD Record* 22(1), 21–27 (1993)
14. OMG, UML 2.1.2 Superstructure Spec., formal/ 2007-11-02 (2007), www.omg.org