Stones Falling in Water: When and How to Restructure a View-Based Relational Database (Extended Version)*

Eladio Domínguez[1], Jorge Lloret[1], Ángel L. Rubio[2], and María A. Zapata[1]

[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. Nowadays, one of the most important problems of software engineering continues to be the maintenance of both databases and applications. It is clear that any method that can reduce the impact that database modifications produce on application programs is valuable for software engineering processes. We have proposed such a method, by means of a database evolution architecture (MeDEA) that makes use of database views. By using views, changes in the structure of the database schema can be delayed until absolutely necessary. However, some conditions oblige modifications to be made. In the present paper we present an approach to detect *when* the restructuring process must be realized and *how* to carry out this restructuring process.

1 Introduction

Software maintenance, and in particular database maintenance, is still one of the major challenges that researchers and practitioners must face in their everyday work. The systematic usage of multi–tier architectures helps to grant independence between applications and their underlying databases, and indeed, to dissociate at least to some degree the maintenance tasks that might be carried out within the distinct levels. Nevertheless, sometimes this dissociation is not feasible and the required modifications affect both applications and databases. It is clear that this process should be done as seamlessly as possible, as is recognized for example in [10] in the case where databases are the artifacts that unleash the process. Liu et al. claim that "if additional functionality can be added [to

* This work has been partially supported by the Ministry of Science and Innovation, project TIN2009-13584, by the Ministry of Industry, Tourism and Commerce, project LISBioBank (TSI-020302-2008-8) and by the Government of Aragon, project LIS (PI108/08).

the database] seamlessly, existing application programs may either optionally ignore it or only require minimal modifications when the added functionality becomes available. Therefore, how to effectively manage the impact of schema modification, clearly, becomes an important issue for achieving such seamlessness". Within this context, any means which delays and minimizes the impact that database changes provoke in the applications is of great value.

Different strategies can be followed to make progress in order to achieve a solution to this issue. In [4], we proposed MeDEA, a Metamodel–based Database Evolution Architecture, which follows a forward maintenance perspective. Roughly speaking, this means that all changes that have to be introduced to the database are issued at the conceptual level. The changes at the conceptual level are, by means of MeDEA, automatically propagated downwards to the logical level and to the extension. As was presented in [4], MeDEA follows a *strict data conversion mechanism*, meaning that conceptual schema changes are immediately propagated to the logical schema and to the extension. However, the strict conversion mechanism may involve a big impact on both the database and the applications. In [5] we improved the approach by proposing a lighter data conversion mechanism that makes use of database *views*. When views are used, the modification of structures is not initially realized (*logical* conversion mechanism), but delayed until necessary, which significantly reduces the impact of required changes. Unfortunately, the use of views does not solve the problem in its entirety, because of the well–known problem of *updatability* of views. It has been proven in the literature [3, 9] that not every DML operation defined on a view can be satisfactorily realized. There exist some structural requisites in order for a view to be fully updatable.

The main goal of the present paper is to show an approach that determines, on the one hand, when a view (generated and inscribed in the context of our evolution architecture) must be restructured in the presence of some particular DML operations, and on the other hand how this restructuring process can be carried out.

The rest of the paper is structured as follows. In the following section we present the basic structures and the way of working of MeDEA, including the running example that we will use in the paper. Section 3 is devoted to outline the settings of the problem we solve. In Section 4 we explain when the database should be restructured and in Section 5 we describe how to do the restructuration. Finally, we present some related works, conclusions and further work.

2 Database Evolution Architecture

The contributions of this paper rely on the MeDEA architecture for database evolution we proposed in [4, 5]. In order to make the paper self–contained, in this section we review the basic ideas of MeDEA such as we described it in previous papers (see [4, 5] for details). At the same time, we introduce some new notions, not included in our prior works, that are used in the subsequent sections.

MeDEA is a metamodel–based database evolution architecture we presented in [4]. MeDEA uses a metamodeling approach for its 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 the DBMS. 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.

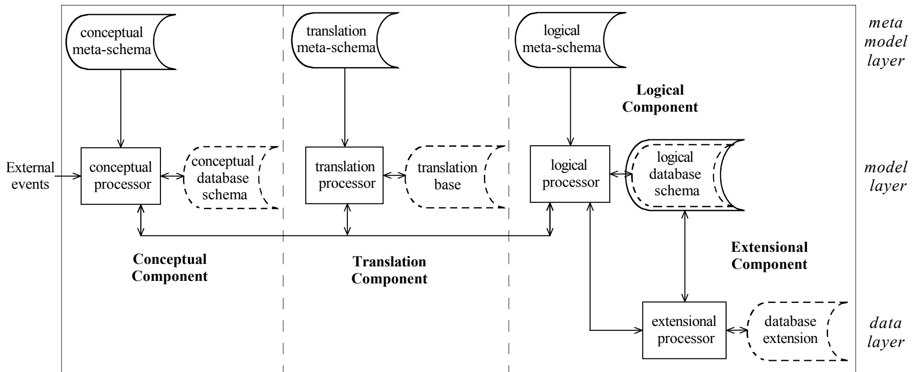

Fig. 1. MeDEA Database Evolution Architecture

2.1 Forward Translation Process

The way of working of MeDEA is as follows: given an EER schema in the conceptual component, the translation algorithm, which is composed of translation rules, is applied to it and creates 1) a set of elementary translations in the translation component, 2) the relational database schema and 3) the extensional database schema. The resultant schemas of this forward translation process have been depicted in the top part of Figure 2.

For instance, as a running example, we will consider an initial EER schema S_0 (see Figure 3, left) with employees and cities. Each employee has an `id`, `name`, `address` and `department` where (s)he works. Each city has an `id` and a

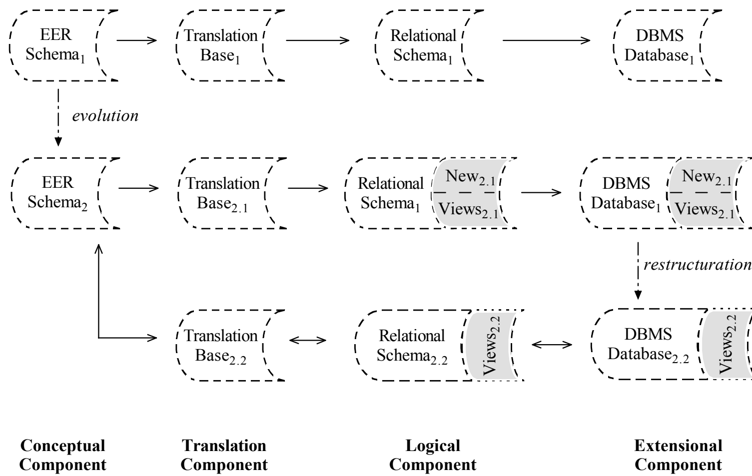

Fig. 2. View–based Evolution Processes within an EER–DBMS setting

`name`. Many employees can live in the same city. From this example, our architecture creates the extensional database schema (see Figure 3, right). Next, the extensional database is populated with data.

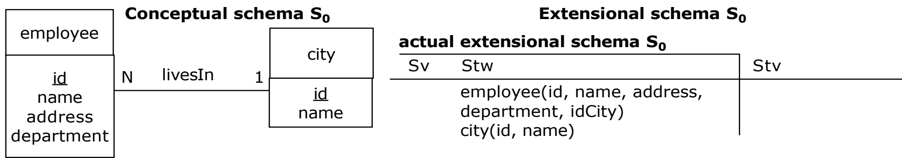

Fig. 3. Initial schema

2.2 Forward View-Based Database Evolution

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. These changes must be propagated to the rest of the components. In order to do this we have proposed two different forward approaches: a strict propagation mechanism [4] and a lazy and logical mechanism [5]. In this paper we assume that the lazy and logical mechanism is applied. According to this proposal, a view–based propagation algorithm is applied which delays the propagation of changes creating views in the logical and extensional levels.

The key aspect of the view-based propagation algorithm is that the old logical and extensional schemas remain unchanged, and the target schemas are not completely created but simulated. In general, when the conceptual evolution

implies the creation of new elements (tables, attributes,...) they are created, but the modification or elimination changes are simulated creating views. This fact is depicted in the central part of Figure 2. It can be seen in this figure that, after a forward evolution propagation process, the relational and extensional schemas include the old schemas, together with the 'Views' piece representing the views that simulate the modification and elimination changes and the 'New' piece representing the added schema elements.

For example, we consider that in schema S_0 the attribute `address` is deleted, giving rise to the conceptual schema S_1 (see Figure 4, left). However the translation of this change does not provoke the elimination of the column `address` in the table `employee`, instead of this, the lazy propagation algorithm delays the elimination of the column creating the view `vEmployee` (see Figure 4, right).

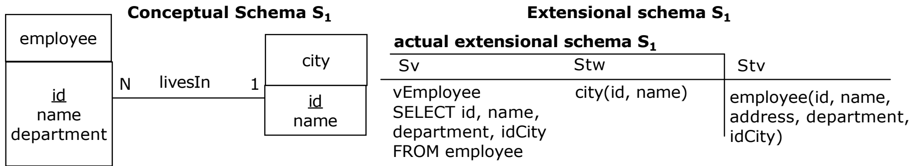

Fig. 4. First evolved schema

In the resultant situation not all extensional elements conform with the new EER schema. For this reason, we are going to introduce several notions (not included in previous papers) in order to classify the database elements making explicit which extensional elements are related with the conceptual ones.

It must be noted that each extensional schema S can be split into two sets: the set of views S_v and the set of tables S_t. In its turn, S_t is split into two sets: the set of tables that have views defined on them, S_{tv}, and the set of tables without any view defined on them, S_{tw}. Just after having translated the conceptual schema to the logical one and before applying any conceptual transformation, the sets S_v and S_{tv} are empty. Moreover, at any time, it is verified that $S_{tv} \cap S_{tw} = \emptyset$. Besides, according to the lazy and logical propagation procedure, only S_v and S_{tw} correspond to conceptual elements, for this reason, we call the set $S_v \cup S_{tw}$ *actual extensional schema*. For example, the partition of the extensional schemas S_0 and S_1 are shown in the right part of Figures 3 and 4.

One of the main advantages of this approach is that, since the extensional database schema remains unchanged, the old software applications can continue being executed directly on the tables where they were defined. In this way the changes on software applications are delayed, minimizing the impact of evolution.

3 Backward-Forward Restructuring Process

Having explained our previous research results on which the contributions of this paper are built, in this section we identify some management problems

that can arise within a view-based relational database and which can imply the restructuration of the database. After explaining these problems, we describe how our prior proposal [4, 5] is enhanced in order to settle them.

Two kinds of DML operations can be performed on the resultant extensional schema: those coming from the old applications and those coming from an external source as, for example, scripts of DML operations. The former do not entail any problem, since, as we have said before, they continue being executed directly on the tables where they were defined, that is tables of S_{tv} and S_{tw}. However the latter, as we will see, can be problematic.

First, we are only interested in external DML operations that conform with the conceptual schema, that is, operations defined on the actual extensional schema ($S_v \cup S_{tw}$). For example, the address of an employee can be modified in the extensional schema S_1 of Figure 4 (it is a valid operation) however we do not intend to consider it because it does not conform with the conceptual schema S_1 (according to this schema this attribute does not exist).

In this context the well-known problem of view updatability arises, that is, DML operations defined on a view cannot always be translated to the base tables. There exist some structural requisites in order for a view to be fully updatable [3, 9].

For example, we are going to consider that a second set of database evolutions occurs. From the schema S_1, the attribute `department` is transformed into an entity type `department`, a new entity type `building` is added and a new relationship type `situatedIn` between `department` and `building` is settled. Finally, a new entity type `project` and a new relationship type `worksIn` between `employee` and `project` are also added. All these changes give rise to schema S_2 and the corresponding extensional schema (see Figure 5).

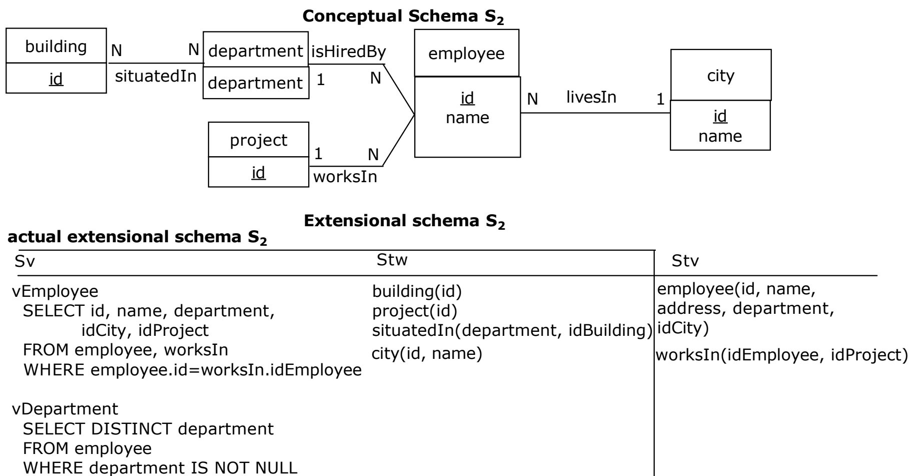

Fig. 5. Second evolved schema

Let us suppose that, according to the conceptual schema S_2, a new department needs to be added. As the base table where the information about departments is stored is the table `employee`, the insertion of the department could be done in this table but we do not consider it a feasible solution, since it would oblige us to insert a new employee for this department (which is not the case). Considering that the entity type `department` is translated into the view `vDepartment`, another option is to insert a department in this view. However, as we will see later on, this view is not insertable and the operation cannot be done. The consequence is that, in this situation, the extensional schema must be restructured propagating the addition of the entity type `department` in order to be able to insert a new department.

As one of our main contributions, we identify in Section 4 *when* an operation on a view can not be translated into base tables. In these cases the extensional, logical and translational schemas must be restructured. In Section 5, as another of our main contributions, we describe a backward–forward maintenance technique for determining *how* these schema restructurations are performed. This restructuration process is depicted in the bottom part of Figure 2. The backward–forward process is represented by means of bidirectional arrows and the modifications performed in the schemas are represented changing the size of the involved pieces, in particular the 'View' piece is represented with a different size and shape.

Before explaining the 'when' and 'how', we want to note that, as we will see later on, the proposed backward–forward process requires that the extensional and logical components obtain information from the conceptual component. For this reason, we have had to introduce a slight modification in MeDEA with regard to the proposal we presented in [4]. The modification is that the arrows representing the communication among the processors are bidirectional.

4 When to Restructure

In this section we answer the following question: Under which conditions is it possible to translate an operation on views into an operation on base tables? This is the updatability problem, which has been widely dealt with in numerous research papers [3, 9] and it is known that not every DML operation on views can be translated to the base tables. The question is to determine which operations on views can be translated into an operation on base tables.

For dealing with this question, we have decided to make independent the definition of updatability of the particular DBMS chosen, so we have defined updatability at the logical level. In this context, we have followed the solution of [9]. The paper [9] offers 'a theory within the framework of the ER approach that characterizes the conditions under which there exist mappings from view updates into updates on the conceptual schema'. It includes a set of definitions and theorems which determine, for entity types and relationship types views, whether they are insertable, deletable or updatable.

Our solution has been to adapt the view updatability algorithm of [9] to our particular context in which views are defined on the logical and on the extensional schema, unlike [9], where views are defined on the E/R schema. This adaptation can be handled because, in our evolution architecture, the conceptual and logical components are interconnected. Thus, we have rewritten the concepts and theorems of [9] in terms of our evolution architecture, generating in this way an integrated solution for the problem of when to restructure the database. For example, about deletability of view relationship types and about insertability in view entity types, [9] offers the following:

'**Theorem 2.** Let R be a view relationship set with relationship derivation $< R_1, ..., R_n >$. R is deletable if and only if R is functionally equivalent to some relationship set R_i w.r.t. $< R_1, ..., R_n >$ where $i \in \{1, 2, ..., n\}$.'

'**Theorem 4.** A view entity type is insertable if and only if the identifier of its base entity type is included in the view'.

In our context, we have reelaborated these theorems as follows:

Deletability Theorem. Let R be a logical select-project-join view defined on base relation schemas $R_1, ..., R_n$, R is deletable if and only if there is a one–to–one correspondence between the rows of R and the rows of R_i for some i.

Insertability Theorem. A logical view V built on a relation schema R is insertable if and only if the attributes of the key of the relation schema R are included in the view V.

If the Insertability Theorem determines that a view is insertable, an insert operation on it is accepted and will be executed provided that the integrity constraints are satisfied. For executing the operation, we will apply an adaptation to our forward database maintenance context of the View Update Translation Algorithm of [9]. Analogously for the Deletability Theorem.

These ideas are gathered in the rule for managing external DML operations (see Figure 6). In this rule, when an operation is accepted, it is executed if the integrity constraints are satisfied. We are going to explain this rule through several examples of application.

```
EVENT: DML operation q on table or view x of Stw or Sv
ACTION:
IF x is table THEN accept q
ELSE
  v <-- getLogicalElement(x)
  IF v is updatable THEN accept q
  ELSE restructure(x);
  END IF
END IF
```

Fig. 6. Rule for managing external DML operations

Example 1. Let us suppose the situation of schema S_1 in which, from the schema S_0, the attribute `address` has been deleted. As a consequence, at the extensional level, we have created a new view `vEmployee` on the previously exist-

ing table `employee`. The view `vEmployee` contains all the columns of `employee` except the column `address`.

Then, if we try to add a new employee through an external DML operation on the view `vEmployee`, the rule of Figure 6 for managing the operations is fired. Next, the view `vEmployee` of the logical level is retrieved and the Insertability Theorem is applied in order to determine whether it is insertable. The view `vEmployee` is built on the relation schema `employee`. The column `id`, which is the key of the relation schema `employee` is included in the view `vEmployee`. So according to our Insertability Theorem, `vEmployee` is insertable and therefore the operation will be accepted.

Example 2. Let us suppose that the situation is that of schema S_2. Then, we try to delete an employee through an external DML operation on the view `vEmployee`. The rule of Figure 6 for managing the operations is fired. Next, the view `vEmployee` of the logical level is retrieved, the Deletability Theorem is applied, which determines that the view `vEmployee` is deletable and the operation is accepted.

Let us now suppose that we try to add a new department through an external DML operation on the view `vDepartment`. Then, the rule of Figure 6 for managing the operations is fired, the corresponding logical view `vDepartment` is retrieved and the Insertability Theorem is applied in order to determine if it is insertable. The view `vDepartment` is built on the relation schema `employee` and the attributes of the key of `employee` are not included in the view vDepartment. So, the view `vDepartment` is not insertable and the operation will not be accepted. In this case, the algorithm determines that the schemas must be restructured in order to insert the new department.

In the next section, we describe the main features of the algorithm which restructures a view–based relational database.

5 How to Restructure

When a DML operation on a view cannot be translated into base tables, the rule for managing DML operations of Figure 6 indicates that the extensional database must be restructured. This section is devoted to explaining how the extensional schema is modified and how the SQL code is generated in order to restructure the extensional database.

The goal of the restructure algorithm is to make the minimal extensional schema changes so that the DML operation can be executed on the new extensional schema. For this goal, we considered two options. One of them was a backward propagation and the other was a backward–forward propagation. With regard to the former, we identified at the extensional level the tables and views that must be restructured (for example, by querying the upper components of the architecture) and the changes were propagated backwards to the logical and to the translation component. For the latter, we proposed to identify, at the conceptual level, the elements which correspond to the view where the operation is defined and to consider reapplying the translation rules to these elements

and to the adjacent elements. These changes will then be propagated forward to the logical and extensional levels. We have chosen the idea of backward–forward propagation for the restructure algorithm because we can reuse the main elements of our architecture and, particularly, the translation component.

Tasks for restructuring. For the backward–forward propagation idea, two tasks are interspersed in the restructure algorithm. First, identify the conceptual elements related with the restructuration. Second, for each one of these conceptual elements its translation to the logical and extensional levels is modified. For the second task, three circunstances can appear:

1. A new translation rule is applied.
2. The same translation rule as before is reapplied but its effects will be different from the effects when it was previously applied.
3. Nothing is changed (the same translation rule would be applied but it would produce the same effect).

As we will see in more detail, these three possibilities appear, for example, when we try to make an insertion on view `vDepartment` of schema S_2. According to the Insertability Theorem, the view is not insertable. So, in order to do the insertion, a new translation rule must be applied to the `department` entity type translating it into a table `department` (first possibility). As a consequence, the entity type `employee` must be retranslated. In this case, as before, it is translated into the view `vEmployee`, using the same translation rule, but the result is different since an attribute related with the new table `department` must be incorporated (second possibility). However, the translation of the entity type `building` must not be changed (third possibility).

Algorithm. After having described the tasks involved, we go into depth about the restructure algorithm by explaining how the two previously described tasks are performed in the algorithm. The input of the algorithm is the view on which the DML operation is defined. First, the conceptual element c which corresponds to the view v is retrieved (line 3). Then, the procedure `considerApplyTranslRule` is executed for the element c (line 4). This procedure, which is used several times in the algorithm, translates a conceptual element performing one of the three possibilities previously mentioned.

If c is an entity type, then the relationship types where c participates may change their translation rules (lines 5 to 12 of Figure 7). For this purpose, the first relationship type processed must be the relationship type which connects c with the entity type which corresponds to the relation schema which contains the information of the view. For example, for the entity type `department`, the entity type which corresponds to the relation schema which contains the information of the view `vDepartment` is `employee`, so the first processed relationship type must be `isHiredBy`.

Then, the reapplication of a translation rule is considered, first for each relationship type (line 7) and, next, for each participant of the relationship type (line 9), with the same three possibilities of translation we have mentioned before. If the conceptual element is a relationship type, then the reapplication of translation rules to its participants is considered (line 15).

```
INPUT: v view
OUTPUT: Changes on the translation, logical and extensional schema
 c<--getConceptualElement(v)
 considerApplyTranslRule(c)
 IF c is entity type THEN
   FOR each r relationship type where c participates DO
     considerApplyTranslRule(r)
     FOR each p participant in r different from c DO
       considerApplyTranslRule(p)
     END FOR
   END FOR
 END IF
 IF c is relationship type THEN
   FOR each p participant in c DO
     considerApplyTranslRule(p)
   END FOR
 END IF
```

Fig. 7. Algorithm for restructuring the translation, logical and extensional schema

A noteworthy characteristic of our algorithm is that the restructuration technique we propose only makes the compulsory changes so that the desired operation can be executed on the new extensional schema. This idea can be metaphorically compared with a stone falling in water. When a stone falls in water, the effect does not remain at the point where the stone touches the water, but it is propagated towards the shore in the form of several concentric waves (and not in an irregular fashion). In a similar way, the retranslation of a conceptual element is the stone which drops onto the schemas and this change generates concentric waves that carry the minimal changes towards other pieces of the schemas.

After having restructured the logical and the extensional schema, there can be queries which are no longer valid, for varied reasons, as they use views which no longer exist. In our example, the queries using the view `vDepartment` are no longer valid. So, these queries must be recreated. As a future work, we want to define a conceptual query language that allows us to specify the queries at the conceptual component. Then, they will be translated by an algorithm to the logical component. Afterwards, when the database is restructured, the queries will be automatically restructured by taking advantage of the translation component. For the moment, the recreation of each query is done manually.

Example. Let us consider again the previously mentioned insert on view `vDepartment` of schema S_2. Then, the rule for managing external DML operations (Figure 6) is fired because the event is an operation on a view of S_v. As, according to the Insertability Theorem, the view `vDepartment` is not insertable, the restructure algorithm is executed. Its input is the view `vDepartment` and its output is the SQL code of Figure 8.

Going step by step in the algorithm, first the conceptual element, that is the entity type `department`, is retrieved (line 3) and the change of its translation rule is executed. To be specific, the first possibility of the previously described possibilities for the translation rules is applied, and the translation rule `EntityTypeToRelSch01` is now executed for the entity type `department`. This application does not modify the conceptual schema (the entity type `department` remains the same) and is captured by the view–based propagation algorithm.

This algorithm modifies values of the architecture components. For example, after the change, the entity type `department` is translated to a relation schema instead of to a view. Next, the logical subalgorithm is executed and, as a result, a new relation schema is created for the `department` entity type. Finally, the extensional subalgorithm applies a rule and a new table `department` is created by means of SQL sentences (a) to (d) of Figure 8. Once the view `vDepartment` is no longer valid, it is dropped (sentence e). From now on, we do not detail the effect of the changes on each component of the architecture and we describe only the effect on the extensional component.

Next, the relationship types where the entity type `department` participates are retrieved in order: `isHiredBy` and `situatedIn`. For the first one, the first case for applying a translation rule is applied, because the relationship type `isHiredBy` can no longer be translated into the view `vEmployee`. This application is captured by the view–based propagation algorithm, which translates them to the rest of the components. In particular, a procedure is fired, which loads data into the column `isHiredBy` just added to the table `employee`. As a result sentences (f), (g) and (h) of Figure 8 are executed. For the participant `employee` of `isHiredBy`, the translation rule `EntityTypeToView01` is reapplied, giving place to sentence (j).

Next, the relationship type `situatedIn` is processed. The sentences (k) to the end are the result of the reapplication of the translation rule `RelSchPerNNRelType` to the relationship type `situatedIn`.

```
a. CREATE TABLE department(id INTEGER, department VARCHAR2(30))
b. INSERT INTO department(department) SELECT * FROM vDepartment
c. UPDATE department SET id=rownum
d. ALTER TABLE department ADD CONSTRAINT pk10 PRIMARY KEY(id)
e. DROP VIEW vDepartment
f. ALTER TABLE employee ADD isHiredBy INTEGER
g. UPDATE employee e SET isHiredBy=(SELECT id FROM department WHERE department=e.department)
h. ALTER TABLE employee ADD CONSTRAINT fk10 FOREIGN KEY (isHiredBy) REFERENCES department(id)
i.  ALTER TABLE employee DROP COLUMN department
j. CREATE OR REPLACE VIEW vEmployee AS SELECT id, name, isHiredBy, idCity, idProject
     FROM employee, worksIn WHERE employee.id=worksIn.idEmployee
k. ALTER TABLE situatedIn ADD idDepartment INTEGER
l. UPDATE situatedIn s SET idDepartment=(SELECT id FROM department WHERE department=s.department)
m. ALTER TABLE situatedIn ADD CONSTRAINT fk11 FOREIGN KEY (idDepartment) REFERENCES department(id)
n. ALTER TABLE situatedIn DROP CONSTRAINT pk5
o. ALTER TABLE situatedIn DROP COLUMN department
p. ALTER TABLE situatedIn ADD CONSTRAINT pk5 PRIMARY KEY (idDepartment, idBuilding)
```

Fig. 8. Generated SQL code

6 Related Work

Within schema evolution, several authors analyze the impact of schema changes on data [2]. However, less attention has been devoted to avoiding or minimizing the impact of schema changes in applications [7]. The few proposals that can be found normally focus their attention on the definition of facilities to perform the reprogramming tasks, for example identifying the impacts [8, 11] or proposing programming techniques to enhance the adaptability of database programs against schema evolution [10]. Another approach, within which our work is included, is to minimize the impact of changes presenting different proposals [6].

The technique we have used to minimize the impact has been carried out by means of views. As a first stage, the changes are gathered by means of views without affecting other application programs [5]. Several authors have proposed the use of views with this aim [13]. However, the main noteworthy characteristic of our proposal is to inscribe it within a model driven development for databases, considering three different abstraction levels: conceptual, logical and extensional and within a metamodeling context.

Other recent papers use ‘some notion of view’ in order to perform database maintenance tasks. For instance, in [1] the starting point is the existence of materialized views that store data integrated from multiple, heterogenous data sources. This marks an important difference with our work, since in our case we do not deal with materialized views but pure relational views that directly grant access to some part of one database schema. Besides, neither an evolution setting, nor an explicit conceptual level are considered in [1]. The paper [12] uses a database conceptual level, by means of a new E/R variant (called EDM) and a conceptual query language (called Entity SQL). However, the concept of view used in that paper is a kind of bridge between queries expressed in Entity SQL and queries expressed in pure SQL, and it does not correspond to the usual notion of relational view that we use in the present paper.

In any case, views do not completely solve the problem due to the well–known problem of view updatability [9]. Several authors have proposed rules to determine when a view is updatable [3, 9]. To deal with this problem, we propose, as the main contribution of this paper, a second stage so that an algorithm detects when an operation obliges a modification of the schema and another algorithm determines only the compulsory schema changes in order to minimize the applications impact. In this way changes are delayed as far as possible.

The view updatability is defined in a general way by several authors whereas we use it in a specific context of schema evolution within a lazy propagation mechanism. In this case we have decided to base our proposal on the one we consider the most complete [9]. But this is defined in ER and we have translated it to the relational model.

Lastly, we highlight that when the changes cannot be delayed any more, views are used to determine the conceptual changes that have to be propagated. For this reason views codify the changes to be undertaken, which is not considered by other authors. Furthermore a DML extensional operation that obliges the extensional schema database to be changed is translated to the most abstract level and then propagated in a forward fashion. The novelty of this proposal is to transform a backward maintenance task [7] into a forward maintenance task.

7 Conclusions and Further Work

The use of views can reduce the impact that database modifications produce on application programs, since changes in the database are delayed until absolutely necessary. However, it can happen that these changes ultimately become mandatory because of the problem of updatability of views. In the present paper, we

have presented an approach that, on the one hand detects *when* such changes convert into compulsory ones, and, on the other hand indicates *how* to perform database maintenance using the tools and facilities provided by the MeDEA architecture. The overall contribution means that database engineers are provided with an infrastructure that allows them to perform semi-automated evolution and maintenance tasks in such a way that the structure and extension of the database is altered only when it is absolutely necessary. This situation allows applications using these modified databases to continue functioning unchanged for a longer time period.

From here, several lines of work are opened up. Our proposal is inspired in part by the work [9], which uses views at the conceptual level. Since we only use views at the logical level, it would be interesting to analyze the impact that the introduction of views at the conceptual level would have on our overall proposal. Another distinct line, but related with the above, would be to introduce a query language at the conceptual level so that updatability problems could be addressed directly at this level.

References

1. S. Chen, X. Zhang and E. A. Rundensteiner, A Compensation–Based Approach for View Maintenance in Distributed Environments, *IEEE Transactions on Knowledge and Data Engineering* 18(8), 1068–1081, 2006.
2. C. A. Curino, H. J. Moon, C. Zaniolo, Graceful Database Schema Evolution: the PRISM Workbench, *Proceedings of the VLDB Endowment*, 1 (1), 761-772, 2008.
3. U. Dayal, P. A. Bernstein, On the Correct Translation of Update Operations on Relational Views, *ACM Transactions on Database Systems* 7(3), 1982, 381-416.
4. E. Domínguez, J. Lloret, A. L. Rubio, M. A. Zapata, MeDEA: A database evolution architecture with traceability, *Data Knowledge Engineering*, 65(3), 2008, 419–441.
5. E. Domínguez, J. Lloret, A. L. Rubio, M. A. Zapata, Model–Driven, View–Based Evolution of Relational Databases, *DEXA'08* , LNCS 5181, 2008, 822-836.
6. J. Henrard, J. M. Hick, P. Thiran, J. L. Hainaut, Strategies for Data Reengineering, *Procs. of the 9th Working Conference on Reverse Engineering*, 211-220, 2002.
7. J. M. Hick, J. L. Hainaut, Database application evolution: A transformational approach, *Data Knowledge Engineering*, 59 (3), 2006, 534–558.
8. A. Karahasanovic, Identifying Impacts of Database Schema Changes on Applications, *Proceedings of the 8th Doctoral Consortium at the CAiSE*, 93-104, 2001
9. T. W. Ling, M.L. Lee, View Update in Entity-Relationship Approach, *Data Knowledge Engineering* 19(2), 1996, 135–169
10. L. Liu, R. Zicari, W. Hürsch, K. J. Lieberherr, The Role of Polymorphic Reuse Mechanisms in Schema Evolution in an Object-Oriented Database, *IEEE Transactions on Knowledge and Data Engineering* 9(1), 50-67, 1997.
11. A. Maule, W. Emmerich and D. S. Rosenblum, Impact Analysis of Database Schema Change, *In 30th Intl. Conf. on Software Engineering*, 451-460, 2008.
12. S. Melnik, A. Adya and P. A. Bernstein, Compiling Mappings to Bridge Applications and Databases, *ACM Transactions on Database Systems* 33(4), 22, 2008.
13. Y. G. Ra, E. A. Rundensteiner, A Transparent Object–Oriented Schema Change Approach Using View Evolution, *Intl. Conf. on Data Engineering*, 165-172, 1995.