Maintaining Schemata Consistency for Interoperable Database Systems

Illarramendi A., Blanco J.M., Mena E., Goñi A., Pérez J.M.

Facultad de Informática, Universidad del País Vasco. Apdo. 649 (20.080) San Sebastián. SPAIN

e-mail: jipileca@si.ehu.es

Abstract

Many interoperable database systems offer the possibility of defining integrated schemata on top of heterogeneous databases. A very important challenge for these interoperable database systems is to maintain the autonomy of the component databases while preserving the correct semantics of the integrated schemata. This paper presents a mechanism that responds automatically to design changes made in component databases which are relevant to one or more integrated schemata. Further, this mechanism provides each component that decides to participate in the interoperable system with the opportunity to choose between assuming the default monitoring provided by the system or customizing it by defining the system responses.

1 Introduction

It is widely recognized that many organizations possess their data stored in distributed, heterogeneous, autonomous data repositories. In order to support coordinated access to this data, interoperable systems have been defined. In the literature we can find many proposals for interoperable systems, and in particular interoperable database systems. In the latter case, there is not a consensus in the number of integrated schemata that they should provide —while some provide a global unique integrated schema of the component schemata (e.g., [1]), others do not provide any global integrated schema at all (e.g., [2]). An intermediate position is supported by those that offer several integrated schemata customized to the specific needs of different applications (e.g., [3] and [4]). Nevertheless, when an integrated schema (global or partial) is provided, the task of maintaining the autonomy of the component database systems, while preserving the correct semantics of the integrated schema, is very relevant.

Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95)
Ed. Tok Wang Ling and Yoshifumi Masunaga
Singapore, April 10-13, 1995

Component autonomy can be addressed from different points of view such as: design, communication and execution autonomy [5]. Design autonomy refers to the ability of component database administrators to choose their own designs with respect to the conceptualization and representation of the data, used constraints, etc. Communication autonomy refers to the ability of a component database to decide whether to communicate with other component databases and lastly, execution autonomy refers to the ability of a component database to execute local operations without interference from external operations and to decide on the order in which to execute them.

In this paper we concentrate on the component's design autonomy. We present a system that responds automatically to design changes made in a component database which are relevant to one or more integrated schemata. The types of design changes that the system monitors are: addition, deletion or modification of data elements that appear in the structural definition of the databases which take part in the interoperation. The three main processors that constitute the system are the *Modifications Detector*, the *Modifications Manager* and the *Consistency Re-establisher*. The joint goal of the two first is to detect and identify, automatically, relevant design changes in component databases, while the goal of the third one is to re-establish automatically the consistency of the integrated schemata whether it is possible or to generate the appropriate warnings to the users.

Among the scarce works that can be found in the literature related to the consistency problems for interoperable database systems we can point out [6], [7] and [8]. The two first are concerned with consistency problems at the instance level, i.e., among data stored in different databases. In [6] a mechanism is presented for specifying mutual consistency requirements —flexible limits within which related data must remain consistent. In [7] a proposal is presented for automatically generating active database rules that maintain consistency in the presence of semantic heterogeneity. Our

system, in contrast, monitors the consistency at the definition level, that is, among the integrated schemata and the autonomous schemata definitions [9]. Therefore, we deal with the important problem of meta data consistency. In [8], a method is proposed that monitors at the definition level too. It sustains the structural semantic integrity of integrated schemata (defined with an object-oriented data model) regardless of the dynamic nature of component schemata. They do not, however, explain how and when the component database modifications will be detected.

Many databases from legacy that can be incorporated into an interoperable database system do not provide facilities for production rules, e.g. prerelational databases. In database systems with such facilities (also known as active database systems [10], [11]) the production rules can be used to monitor the consistency problems. Others provide them but with some constraints, e.g. RDB^{TM} supports only a concrete type of system response. For this reason, it is necessary to define a new mechanism for interoperable database systems, independent of the component systems, to solve the problem of maintaining the consistency.

Another feature of our approach is that, when a new database becomes part of the interoperable database system, it allows one to select the default monitoring provided by the system or to customize the monitoring defining the system responses. Moreover, notice that our system exempts the component database administrators from the responsibility of informing the interoperable system every time a design change is made.

Finally, in the implementation of the system the Client/Server approach and object-oriented techniques have been used. The Client/Server approach allows us a parallel processing of different design changes. Furthermore, object-oriented techniques permit the encapsulation of the peculiarities of the data structures provided by the catalogs used by the different component database management systems, and so the access is made to the different catalogs by an interface. Notice that the previous techniques facilitate the extensibility of the system to other types of interoperable system. In the remainder of this paper, first we present briefly the global interoperable database system architecture, then we explain the details of each component that is concerned with the mechanism to maintain schemata consistency, and last we introduce a motivating example.

2 System Architecture

In general, an integrated schema is generated through a process in which first of all, correspondences among different *objects*[1] of distinct databases are expressed and then some integration rules are applied. Usually, the databases that must be integrated are heterogeneous so a previous step is needed where the heterogeneity due to the use of different data models is eliminated. Notice that not all objects of the components databases must belong to the integrated schema. Each component database system has the autonomy to decide which objects will be exported to the interoperable system. Nevertheless, when a change occurs in a component database, for example an object is deleted, that is relevant to an integrated schema, that change must be propagated to the corresponding integrated schema definition, because otherwise this definition would remain inconsistent. In general, more than one integrated schema could be affected by a change in a component database.

Our proposal of an architecture for an interoperable database system contains four main modules, *Translator, Integrator, Query-Processor,* and *Monitor.* We first present briefly the three first and then we explain in more detail the last one, which is concerned with the mechanism to maintain schemata consistency.

- Translator Module. This produces, with the help of the Person Responsible for the Integration (PRI), a new semantically richer schema definition from a conceptual schema of a component database.
- Integrator Module. This produces an integrated schema by integrating a set of schemata previously obtained by the Translator Module.
- Query-Processor Module. This obtains the answer to the user queries over the integrated schema by accessing the component databases. The Query-Processor Module has two kinds of components, the Global Query Processor and the Local Query Processor.
- Monitor Module. This responds automatically to design changes made in component databases which are relevant to one or more integrated schemata.

Concentrating in the last module it constains four main components, three processors, the *Modifications Detector*, the *Modifications Manager* and the *Consistency Re-establisher*, and a new catalog, the *System Consistency Catalog*, that contains the relevant catalog information (see figure 1). What we mean by *relevant* is that part of the database(s) catalog information

[1] Object here refers to data elements that appear in the structural definition of the databases, relations and attributes for the relational model; data items, records and sets for the network, integrity constraints, etc.

which the *Modifications Manager* needs to discover the modifications.

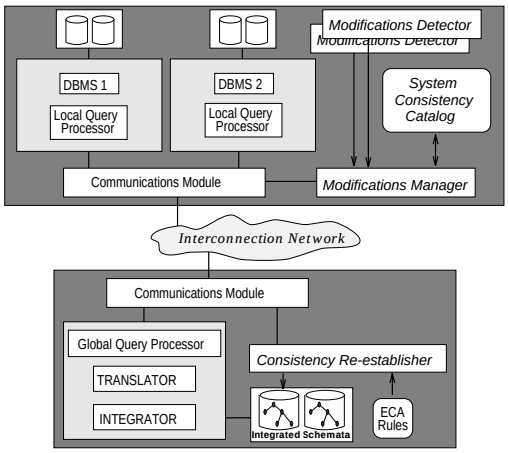

Figure 1: Operational System Architecture

Three main steps are followed by the Monitor Module to achieve its goal:

1. When a design change is made in a component database, the Modifications Detector processor detects the situation and sends a message to the Modifications Manager.

2. Then, the Modifications Manager analyses the message and discovers if the modification is relevant, in such cases, it generates one or more events. During the discovery process, the Modifications Manager uses the information stored in the System Consistency Catalog.

3. The Consistency Re-establisher detects the events and tries to re-establish the integrated schemata consistency by using a set of rules.

From an operational point of view, a Client/Server architecture has been used. There is a Client application, the Modifications Manager, and one (or more) Modifications Detector processor (one for each DBMS) at each node to monitor the schemata modifications of the databases defined in that node that belong to the interoperable system. There also exists a Consistency Re-establisher application, for each node where integrated schemata are defined, that behaves as a server application of the Modifications Managers. This type of architecture allows us to define the parallel processing of different design changes, that is, different Modifications Managers (probably, in different nodes) could invoke at the same time the Consistency Reestablisher; the Client/Server approach guarantees the concurrency in the answer to the clients, duplicating the server if it is necessary. Alternatively, it could be possible to have only one Modifications Manager for the whole system and so one Consistency Re-establisher. The best organization could be chosen according to the requirements of the organization.

3 The Modifications Detector

The goal of the Modifications Detector is to detect changes that occur in a database catalog. To detect if a database design modification has occurred from outside the DBMS (DataBase Management System) is not a simple task. To do it, three different solutions are possible:

- To associate actions with the definition of objects that must be monitored, for example to attributes and tables in a relational database.

- To define an interface on top of the DBMS and to force the component database administrator to use this interface every time that a database design modification is to be made.

- To detect the modification with the help of the Operating System. DBMSs store the catalog information in files. Therefore, every time that the database administrator makes a design change, the corresponding catalog file is modified. Using the operating system functions, modifications of files can be detected. Note that it is not necessary to access the files, but only to the directory information to obtain the date of their last modification. For each DBMS it is necessary to know the name of its catalog files and their addresses.

The first solution is only applicable to systems that provide production rules and this is not always the case for the databases that must be integrated, e.g hierarchical databases. The second one, although general, would go against the maintaining of component database autonomy. The third solution eliminates the limitations of the two first, it does not require the component database system to be active, neither interfere in its autonomy, therefore, we apply the third solution. For that we have defined a *demon*, that is, a process that monitors catalog file updates. A demon can be activated with a certain time granularity. A different demon exists for each different DBMS. However, two databases defined using the same type of DBMS are monitored by two instances of the same type of demon (see figure 2).

Depending on the DBMS catalog files can be updated every time that the modification occurs only in the intensional part, or every time that a modification

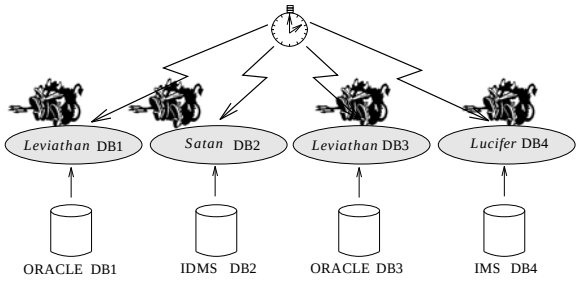

Figure 2: Monitoring different databases

occurs in the intensional part as well as in the extensional part of a database (i.e., when the extensional and the intensional parts are stored in the same file). For the last case, the time granularity associated with the demons will be greater because in this way the detection of many irrelevant changes is avoided.

In general, the time granularity will be decided by the PRI (although a default one is provided by the system), for example one day, taking into account that design changes do not occur very often. Moreover, notice that in this case the *off-line* process of detecting changes could be executed when the system is not overloaded. Nevertheless, this type of behaviour could admit, during a period of time, some inconsistencies among component and integrated schemata. Furthermore, observe that changes that occur while the demon is not active are treated together. This has the benefit of avoiding useless actions in situations such as when an object is deleted from a database and then it is again created.

The abstract specifications of all types of demons are equivalent, they only differ in the catalog files associated with each type of DBMS that they must monitor. The algorithm for the demon that watches over IDMS^{TM} databases is shown in figure 3.

```
Input Arguments : Dbname Dbpath Dbowner Dbuser Dates

(* The four first arguments are used to identify the database part
   that must be monitored. The last argument contains the dates
   associated with the database files that must be monitored *)

   obtain_names_of_database_files("IDMS");
   flag = 0;
   for each database file do
        obtain_new_file_name(Dbpath);
        obtain_date;
        if new_date ≠ old_date then
             flag = 1;
             update_date(Dates,new_date);
   If flag then (* a modification has happened *)
        warning_mod_manager(Dbname,Dbpath,IDMS,owner,user);
        update_dates(Dates);
```

Figure 3: IDMS^{TM} demon algorithm

4 The Modifications Manager

The Modifications Manager's goal is to identify the design changes made in a component database which is relevant to integrated schemata definitions. Four steps are followed by this processor (see figure 4):

1. It receives a message from the Modifications Detector with the identification of the concrete component database where a change has been detected.

2. Then, it sends a request to the database in which the modification has been detected, asking about the catalog information associated with it that is relevant for the integrated schemata.

3. Later, it compares the information returned by the database with the information associated with the same database in the System Consistency Catalog, and discovers if the modification is relevant and, in such cases, the type of modification.

4. Finally, it generates an *event* (or *events*) for the relevant modification which the Consistency Reestablisher can detect and handle.

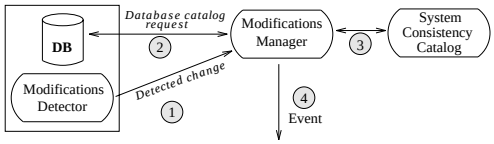

Figure 4: The Modifications Manager Processor

Concerning the first step, there is no problem with various simultaneous calls of the Modifications Detectors, because the use in the implementation of the Client/Server architecture allows that the Modifications Manager will be duplicated for attending each call. This feature permits a parallel process whenever it is necessary.

With respect to the second step, in order to define a solution that could permit access to the different catalogs in a uniform way, object-oriented techniques have been used to provide a view of each component database catalog. This view permits us to encapsulate the peculiarities of the data structures provided by the catalogs used by the different component database management systems. Since the view is represented in an object-oriented data model, its implementation requires the writing of a set of operations (*methods*), which implement the interfaces of the classes in the view in terms of the primitives provided over the component catalogs. For example, the object-oriented view of a catalog supported by a rela-

tional database management system is constituted by the following class[2]:

class relational_database

inherit database

features

obtain-tables is deferred

obtain-attributes is deferred

end – end class relational_database

The operations *obtain-tables* and *obtain-attributes* are primitive operations defined in terms of the specific languages provided by the component database management systems. The implementation of primitive operation *obtain-tables* when dealing with an Oracle^{TM} relational database management system is shown in figure 5.

```
class oracle_relational_database
 inherit relational_database
 features
 obtain-tables is
    EXEC SQL CONNECT :username IDENTIFIED BY :password;
    result = create_table(); /* the resultant table is created */
    result = add_column(result, "CHAR", "NAME");
    result = add_column(result, "CHAR", "TYPE");
    EXEC SQL
    DECLARE tables CURSOR FOR
        SELECT TABLE_NAME,TABLE_TYPE
        FROM ACCESSIBLE_TABLES
        WHERE OWNER <> 'SYSTEM' AND OWNER <> 'SYS';
    EXEC SQL OPEN tables;
    do
    {   EXEC SQL FETCH tables INTO nam_tab, typ_tab;
        if (sqlca.sqlcode == 0)
        {   tuple = create_tuple();
            tuple = add_value(tuple, nam_tab);
            tuple = add_value(tuple, typ_tab);
            result = add_tuple(result, tuple);}}
    while (sqlca.sqlcode == 0);
    EXEC SQL CLOSE tables;
    EXEC SQL COMMIT WORK RELEASE
    return(result);
```

Figure 5: *Obtain-tables* operation for Oracle^{TM}

In the third step, a comparison is made between the information stored in the catalog of the modified database that is relevant for an integrated schema and the information stored for that database in the System Consistency Catalog. This comparation permits the identification of relevant changes.

The *System Consistency Catalog* contains the relevant catalog information before the last change. For each component database of the interoperable system, this catalog contains the names of the nodes where exist integrated schemata to which it is related and the design information that must be monitored (see figure 6).

This information is created during the process of generating integrated schemata and is updated by the

[2] The used notation is taken from Eiffel.

View identifier (DBname, DBpath, DBMS, DBnode, owner, access-user)	Db1, /soft/db/db1, IDMS FD, root, pri	
Nodes	sunsys	
Sets (owner, member)	Can_be_manufactured (Product, Employee)	
Records	Product	Employee
Data Items	P# integer 8 Name char 20 Price integer 10 Tax integer 6	E# integer 8 E_name char 18 Address char 30

View identifier (DBname, DBpath, DBMS, DBnode, owner, access-user)	Db2, /usr/oracle, ORACLE DD, root, pri	
Nodes	sunsys, vax2	
Tables	R_products	Orders
Columns	P# integer 10 Namep char 15 Descr char 50	O# integer 9 P# integer 10 Items integer 3
Indexes	unique P#	unique O#

Figure 6: Part of the System Consistency Catalog

CHANGE TYPES	*OBSERVED EFFECTS*
Addition/deletion of tables	ADD-TABLE *table* DROP-TABLE *table*
Addition/deletion of columns	ADD-COL *column, table* DROP-COL *column, table*
Columns modifications	TYPE-MOD *col,tab,oldtype,new* LENGTH-MOD *col,tab,oldlength,new*
Addition/deletion of indexes	ADD-IDX *index, type, columns* DROP-IDX *index, type, columns*
Addition/deletion of constraints	ADD-CONSTR *constr_type, def* DROP-CONSTR *constr_type, def*

Figure 7: Types of changes in relational databases

Modifications Manager when a relevant change has been identified. Although some of this information is redundant (it appears in the catalogs of the component databases) it is necessary to store it in order to have a partial image of the situation before a new change takes place.

Finally, in the fourth step, the generated events for the relevant changes are of the following type:

schema modified on schema_name (list of changes)

e.g. *schema modified on schema_two*

(DROP-TABLE R_PRODUCTS)

A list of relevant changes is associated as parameter with each event[3]. These changes in the case of monitoring relational databases are shown in the figure 7.

Notice, that the enumerated changes correspond to effects observed in the database catalog and not to the real operations. For example, the change *DROP-COL* does not necessarily mean that the sentence DROP-COL has been used, however the effect of deleting a column can occur when the view that is exported to

[3] Events can be parameterized such that information can be passed to the condition or action parts, if necessary [12].

the interoperable system is redefined.

5 The Consistency Re-establisher

The Consistency Re-establisher's goal is to try to re-establish integrated schemata consistency according to the relevant changes made in a component database. It manages a set of rules defined in terms of ECA rules [13]. An ECA rule *has an event that triggers the rule, a condition describing a given situation, and an action to be performed if the condition is met* [14]. The general format of an ECA rule in our context is

When *schema modified on DB_name*

If *condition*

Do *action*

As can be seen, the type of event considered is *schema modified on DB_name*, that is, a relevant schema modification has occurred in a concrete database. Having only one type of event can seem pretentious to deal with event condition action rules. The reason for our selection is the following: our global idea is to incorporate a mechanism to interoperable database systems that monitors consistency at different levels. At the schema level, that is among component databases schemata definitions and the integrated schemata definitions, at the instance level, that is among the properties about the extension of the component databases that have been used for generating the integrated schemata and even at the application level when the applications can be also integrated. Therefore, three different types of events could be treated that could have associated distinct set of rules. Moreover, these event types could be specialized in future refinements of the mechanism, if necessary.

Example of schema level ECA rule:

When *schema modified on DB_name*

If *length-mod table,column,old,new*

Do *modify affected roles*

Example of instance-level ECA rule:

When *extension property modified on DB_name*

If *functional property does not hold*

Do *group objects*

In figure 8 a triggering of this instance-level ECA rule is shown.

Example of application-level ECA rule:

When *application modified over DB_name*

If *change on preconditions of x*

Do *check depending-applications(x,newprecond)*

[4] Functional dependencies can be used by the Translator Module to obtain a semantically richer schema.

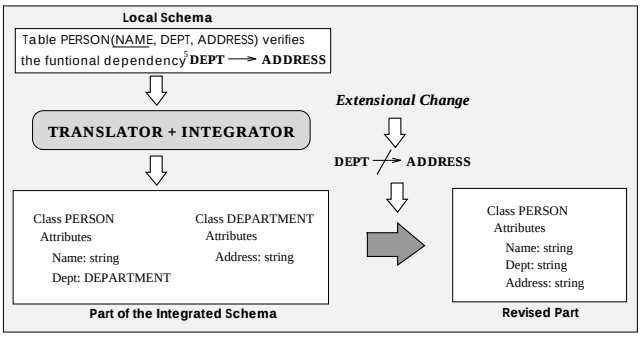

Figure 8: Triggering an instance-level ECA rule

In general, rules are generated automatically when the integrated schemata are created and so links among the integrated schemata and the underlying databases are established. However, rules can also be defined or customized by the PRI.

5.1 Schema-level rules

Now, we only concentrate on schema-level ECA rules. The condition part of schema level type of rules can be classified into three families: addition, deletion, or modification. Inside each family, two different types of conditions are considered, one that corresponds to generic formulations such as *ADD X* where *X* can be a table, a record, an attribute, an index and so on; and other that corresponds to concrete formulations such as ADD *Table Product*. Furthermore, the action part can also be classified into three different groups:

- Pre-defined operations that permit the automatic re-establishment of the consistency of the integrated schema.
- Warnings to the users of the integrated schema. Warnings are generated in the case that an automatic re-establishment is not possible. The warnings will appear every time that a new user wants to work with an inconsistent integrated schema and only will disappear if the PRI eliminates them explicitly.
- Calls to specific procedures defined by the PRI. These specific procedures allow the PRI to customize the behaviour of the Consistency Re-establisher.

In the following we present the different types of rules.

Addition rules

The addition rules usually will have warnings in their action part. When a new object is defined in

a component database that must be incorporated into the integrated schema, this requires then that first it is represented using a canonical model provided by the interoperable system, and second, that the right place in the integrated schema for it can be identified. Many times both steps cannot be performed automatically.

When *schema modified on DB1*

If *ADD record*

Do *Warning ('Call to the Integrator Processor')*

Rules with calls to specific procedures will be defined by the PRI in certain situations.

Deletion rules

The deletion rules can have any kind of action part. A pre-defined DELETE operation for removing an object that belongs to an integrated schema when its support is eliminated from the component database.

When *schema modified on DB1*

If *delete record Product*

Do *DELETE PRODUCT from the Integrated Schema*

A warning when the deletion of an object from the integrated schema could produce the eliminations of other relevant objects (*cascades delete*). Last, call to a specific procedure when it is required a concrete established behaviour.

Modification rules

The modification rules will allow automatic re-establishments and hence will have pre-defined operations in their action part. Nevertheless, particular modifications will require to send warnings to the users.

When *schema modified on DB1*

If *type-mod table,column,number,char*

Do *assign a transformation function*

5.2 Some Implementation Features

From an operational point of view, when an event arises (in our context a relevant component database modification has occurred) the *Consistency Re-establisher* detects it and triggers a set of rules. However, in many situations it could happen that this set is large and so, in order to improve the performance, we have incorporated indexes. The first index corresponds to the type of modification and the second one to the name of the basic elements (see figure 9).

6 Motivating Example

We present a situation where two different databases (actually simplified versions of them), one defined for

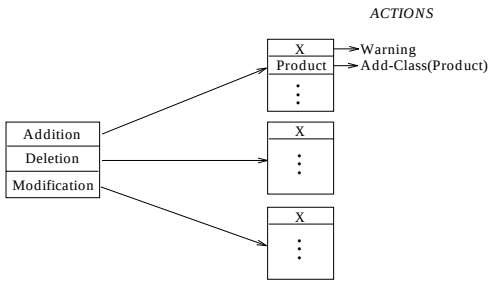

Figure 9: Defined Indexes

the financial department (FD) of a company using a network DBMS and another one defined for the design department (DD) of the same company using a relational DBMS, take part of an interoperable system (see figure 10).

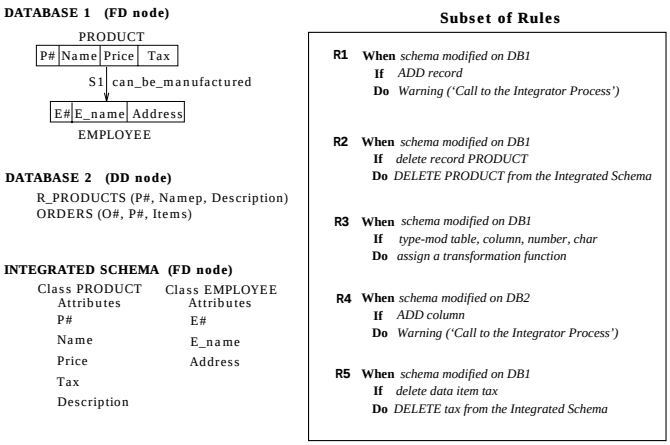

Figure 10: Example of an interoperable DB system

Three different situations are shown:

- Suppose that products are classified into families in the design department and so a new attribute *family* in the table *R_PRODUCTS* of DATABASE2 is introduced. This modification would be first detected by the Modifications Detector, and then sent to the Modifications Manager which would generate an event. The Consistency Re-establisher would detect the event and as a result the R4 rule would be activated. The warning will appear to the PRI who can decide if the new attribute is relevant for the integrated schema and if so, what steps are needed to integrate it.
- The attribute *Namep* of the table *R_PRODUCTS* in DATABASE2 is deleted because in the design department decide that they are not useful any more. This time the Modifications Detector would detect the modification. However, the

Modification Manager would classify it as irrelevant because it has not an image in the integrated schema. Notice that the attribute *Namep* is not exported to the integrated schema.

- The exported schema from the DATABASE1 is redefined eliminating the data item *tax* of the record *PRODUCT* because, for example, a new law eliminates taxes for all the products that the company manufactures. This time, the modification would be detected, identified as relevant, and generated an event that the Consistency Re-establisher would detect and the R5 rule would be activated that permits an automatic re-establisment.

7 Conclusions

In this paper, a system that responds automatically to design changes made in component databases, that are relevant to one or more integrated schemata has been discussed. To our knowledge, this work is the first to address in detail a method to maintain schemata consistency for interoperable database systems. The system described here solves a real problem and gets to enlarge the autonomy concept in the interoperable database system context.

The system allows one to choose between accepting a default monitoring, or customizing the monitoring by defining the system responses. It relies on the use of event condition action rules. These type of rules use event driven invocation of actions.

Furthermore, the system is applicable not only to interoperable database systems, but its funtionalities can be easily extended to other types of interoperable systems. This last property is due to the actual use of object oriented techniques for the system implementation that permit encapsulation of the peculiarities of the local systems that belong to the interoperable system.

Acknowledgements

We like to thank N. Paton, and O. Díaz for their valuable comments.

References

[1] C. Collet, M. N. Huhns, and W. Shen. Resource integration using a large knowledge base in CARNOT. *IEEE Computer*, December 1991.

[2] W. Litwin. An overview of the multidatabase system MRDSM. In *Proceedings ACM National Conference*, 1985.

[3] E. Bertino, M. Negri, G. Pelagatti, and L. Sbatella. Integration of heterogeneous database applications through an object-oriented interface. *Information Systems*, 14(5), 1989.

[4] J.M. Blanco, A. Illarramendi, and A. Goñi. Building a federated database system: an approach using a knowledge based system. To appear in the Int. Journal on Intelligent and Cooperative Information Systems., 1995.

[5] J. Veijalainen and . Popescu-Zeletin. Multidatabase systems in ISO/OSI environment. In *Standards in Information Technology and Industrial Control. Malagardis N. and Williams T. Eds. North Holland*, 1988.

[6] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *Computer. December*, 24(12), 1991.

[7] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proc. of the VLDB 93.*, 1993.

[8] W. Sull and R. L. Kashyap. A self-organizing knowledge representation scheme for extensible heterogeneous information environment. *IEEE Transactions on Knowledge and Data Engineering*, 4(2), April 1992.

[9] J. M. Blanco, A. Illarramendi, J. M. Pérez, and A. Goñi. Making a federated database system active. In *Database and Expert Systems Applications*. Springer-Verlag, 1992.

[10] U. Dayal. Active database management systems. In *Proc. of the 3rd Int. Conf. on Data and Knowledge Bases*, November 1988.

[11] U. Dayal, A. Buchmann, and D.R. Mc.Carthy. Rules are objects too: A knowledge model for an active object-oriented database system; dittrich k.r.(ed.). In *Proc. 2nd Int. Workshop on Object-Oriented Database Systems, Lecture Notes in Computer Science. Springer*, 1988.

[12] S. Gatziu and K.R. Dittrich. Events in an active object-oriented database system. In *Proc. of the 1st International Workshop on Rules in Database Systems. Edimburg, U.K.*, 1993.

[13] K.R. . Dittrich and U. Dayal. Active database systems. In *Proc. of the VLDB 91. Tutorial Notes*, 1991.

[14] O. Diaz, P.M.D Gray, and N.W. Paton. Rule management in object-oriented databases: A uniform approach. In *Proc. of the VLDB 91.*, 1991.