Monitoring the Evolution of Databases in Federated Relational Database Systems

Alfredo Goñi† Arantza Illarramendi‡ Eduardo Mena*‡ José Miguel Blanco‡

† Departamento de Informática e Ingeniería de Sistemas, Universidad de Zaragoza.
María de Luna, 3. 50015 Zaragoza, Spain.
e-mail: agoni@prometeo.cps.unizar.es, phone: + 34 76 762105, fax: + 34 76 762111

‡ Facultad de Informática, Universidad del País Vasco.
Apdo. 649, 20.080 Donostia-San Sebastián, Spain.
e-mail: jipileca@si.ehu.es, phone: + 34 43 218000, fax: + 34 43 219306

Abstract

Federated Database Systems allow users to work with data stored in multiple databases by formulating queries over an integrated view. This kind of systems are of special interest for the emerging type of information systems. In this paper we present an active mechanism to keep consistency in Federated Database Systems. Event-Condition-Action rules are used as the knowledge model. First, we show the types of events and the features of the event generator. These type of events are related to changes on both, the extension and the intension of the component databases. The event generator uses facilities provided by the operating system as well as available active mechanisms supported by the database management systems that take part of the federation. Second, we describe the type of conditions that appear in this context and show the degree of difficulty that entails verifying some of them. Third, we explain the type of actions considered.

1 Introduction

It is of special interest for many companies to work, in a uniform way, with data stored in multiple databases. MultiDataBase Systems (MDBMS) provide one solution to that requirement allowing to make operations that imply more than one database system, each one might be centralized or distributed. Federated DataBase Systems (FDBS) are a special type of MDBS where an integrated view is provided[1]. This integrated view is the result of an integration process among the schemata of the pre-existing autonomous databases. There exist many distinct approaches for building a FDBS, namely the Entity-Relationship model approach [LNE89, SPD92], the Object-Oriented approach [ASD$^+$91, CT91], and the Knowledge Representation Systems (KRS) approach [CHS91, SGN93]. In our case we have built a FDBS that integrates several heterogeneous relational databases[2] by using a particular type of KRS based on Description Logics[3] (DL system). If we do not consider yet the component that monitors system consistency then three main components can be distinguished in our FDBS, namely: Translator, Integrator and Query Processor.

In figure 1 the architecture of the FDBS is shown.

*The work of this author was supported by a grant of the Government of the Basque Country.

[1]Other authors consider that only some federated databases provide an integrated view [BG92]; they call them *tightly coupled federated databases.*

[2]Although they all use the same data model, semantic heterogeneity still can remain.

[3]Also known as Terminological Logics or based on KL-ONE [BBMR89, PSKQ89].

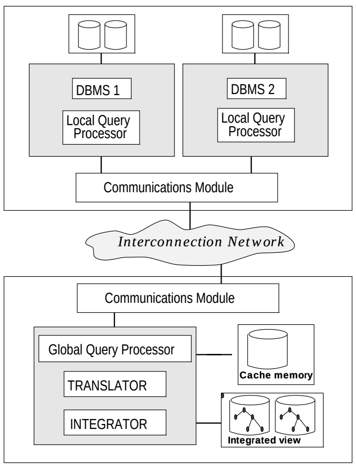

Figure 1: Architecture of the FDBS.

- The *Translator* component produces a knowledge base from a conceptual schema (or a subset of it, called exported schema) of a component database. The resultant knowledge base is semantically richer than the relational database schema, therefore this component has to capture, with the Person Responsible for the Integration (PRI)'s help, the semantics that is not expressed explicitly. Information about dependencies (inclusion, exclusion and functional dependencies), null values and semantic properties (domain information for attribute values) are used by this component.

- The *Integrator* component produces an integrated view by merging a set of knowledge bases previously obtained by the Translator component ([BIG94]). This integrated view is also represented as a knowledge base whose extension is in the component databases. During the integration process a set of correspondences between data elements[4] of the knowledge bases that must be integrated is defined by the Person Responsible for the Integration (PRI) and new ones can also be deduced by the system. The types of correspondence between data elements are: equivalence, inclusion, overlapping and disjointment.

- The *Query Processor* component, formed by the *Global Query Processor* and the *Local Query Processors,* obtains the answer to a user query formulated over the integrated view by accessing the databases or a *cache memory.* The query processing is more efficient if some data are cached in order to avoid accessing the component databases every time a user query is made ([GIMB97]). The Global Query Processor optimizes knowledge base queries, establishes which databases contain the requested information, decomposes a query into subqueries that will run over the component databases and reconstructs the answer from the different results obtained for the subqueries. Local Query Processors make local optimizations and find the answer for the subqueries.

For the implementation of our FDBS we have used the Client/Server approach in such a way that there exists a Client application dealing with the integrated view and several Server applications, one for each database that participates in the FDBS.

In general in any FDBS, the task of maintaining the autonomy of the component database systems, while preserving the correct semantics of the integrated view and the consistency of the cache memory, is very relevant. So, a need arises for an active mechanism that

[4]Classes and attributes in our case.

- monitors changes on the schema definitions of the databases that take part of the federation, that affect the integrated view definition;
- monitors changes on the extension of the databases that take part of the federation in order to discover situations that:
  - invalidate some database dependencies that were used when building the integrated view. The considered dependencies are: inclusion, exclusion and functional;
  - invalidate some correspondences defined among classes and/or attributes. These correspondences were also used when building the integrated view;
  - make inconsistent the set of data cached within the integrated view. Some data of the databases are cached as instances of classes (cached classes) or as values of attributes (cached attributes) that belong to the integrated view.

Among the works that can be found in the literature related to the consistency problems for Multidatabase Systems we can point out [RSK91, CW93, SK93, SK92]. The first three references are concerned with consistency problems at the extensional level, i.e., among data stored in different databases. In [RSK91] a mechanism is shown to *specify* mutual consistency requirements –flexible limits within which related data must remain consistent–. In [CW93] a proposal is presented to automatically *generate* active database rules that maintain consistency in the presence of semantic heterogeneity. In [SK93] a proposal is described to automatically generate rules and constraints from declarative specifications that express *approximate* consistency requirements. In the fourth one, [SK92], a method is proposed that monitors at the intensional level. It sustains the structural semantic integrity of integrated views (defined with an object-oriented data model) regardless of the dynamic nature of component schemata. However, it is not explained how and when the component database modifications will be detected. To our knowledge our work is the first that considers both, intensional as well as extensional level changes. Those changes can affect the consistency of the integrated view and of the cache memory. Notice that the only monitored changes are *bottom-up* changes, that is, those produced on the component databases that affect the integrated view or the cache memory. We do not allow to change *directly* the definition or the extension of the integrated view by using the knowledge representations system.

The goal of this paper is to present an active mechanism to keep consistency of the integrated view and the cache memory of our FDBS with respect to the component databases, using Event-Condition-Action (ECA) rules as the knowledge model [DBM88]. An ECA rule has an *event* part that triggers the rule when such an event happens, a *condition* part that describes a situation and an *action* part that has to be executed when the condition is satisfied. We show first the type of events generated and the features of the event generator. Next, we describe the type of conditions that appear in this context. Third, we explain the type of actions considered that try to re-establish the consistency.

2 Events: Types, Detection and Generation

In our context, events are generated when changes occur in the definition or in the extension of the component databases. Those changes can be produced by either the component database administrators or application programs.

2.1 Event types

Two types of changes can affect the consistency of the integrated view and the cache memory: 1) *intensional changes*, i.e., changes in some database schema, and 2) *extensional changes*, i.e., changes in tuples of a relational table in a component database. However, not all the intensional or extensional changes affect the integrated view or the cache memory. For this reason, it is necessary to select, among all changes, only the relevant ones. This distinction between changes and relevant changes signals the existence of two levels of events:

- *low-level* events, related to intensional and extensional changes. Those events are generated on the component databases by the database administrators or application programs, and
- *high-level* events, related to *relevant* intensional changes or extensional changes in tables that form part of the *mapping information*[5] associated to classes or attributes of the integrated view. Those events are generated by the event generator after *low-level* events have occurred.

Only high-level events are used in the ECA rules. The reason to distinguish between low-level and high-level events is to reduce the number of activations of ECA rules. The different high-level events are:

- *Schema modified on DB_name.* This event type signals an intensional change that affects the consistency of the integrated view.
- *Extension modified on table_name T*, that signals an extensional change that affects the consistency of the integrated view if a database dependency defined over a table T becomes false.
- *Extension affected on class_name C.* This event type signals an extensional change that affects the consistency of the integrated view if a defined correspondence among classes becomes false or if a cached class becomes inconsistent.
- *Extension affected on an attribute A of a class C.* This event type signals an extensional change that affects the consistency of the integrated view if a defined correspondence among attributes becomes false or if a cached attribute becomes inconsistent.

For all the previous high-level events, the list of relevant changes is associated as a parameter[6].

2.2 Event Detection and Generation

In order to *generate* high-level events, low-level events must be *detected* first. In the following we present both of them, that may be related to intensional or extensional changes.

- Low-level events related to intensional changes.

  Information about component database schemata is stored in different catalogs. It is in those catalogs where the intensional changes are reflected. We find that many relational database management systems do not permit the definition of triggers for the catalog tables. This means that it is necessary to define a new processor to detect intensional changes. This processor must not interfere in the autonomy of the component database. In our case we have defined a processor called *Modification Detector* that monitors updates in the catalog files by using operating system functions [IBM$^+$95]. In fact, it is implemented as a *demon* that periodically looks up the date of the catalog files. A different demon exists for each distinct DBMS. However, two databases defined using the same DBMS are monitored by two instances of the same demon.

[5]The *mapping information* relates the classes and attributes of the integrated view with the tables and attributes of the component databases and permits the identification and retrieval of their corresponding instances and attributes values in order to answer the user queries.

[6]Events can be parameterized so that information is passed to the condition or action part if necessary [GD93].

- Low-level events related to extensional changes.

  Taking into account that it is getting more usual that relational DBMS offer the possibility of defining triggers over user tables then it is possible to define a trigger over every table for which the extension must be monitored. All the insertions, deletions or modifications in those tables are written in an *incremental* table that stores the changes. As a first optimization, non-interesting changes in the incremental table can be removed (for example insertion and then deletion of the same tuple).

- High-level events related to intensional changes.

  For the task of identifying, among all the detected intensional changes, only the relevant ones and therefore generating the corresponding high-level events (events of type *Schema modified on DB_name*), it is necessary to define another processor. In our case we have defined the *Modification Manager*. This processor identifies relevant intensional changes by comparing the catalog of the database where the change has been detected with the *System Consistency Catalog*[7] to see if the definition of some table or attribute that form part of the mapping information corresponding to some class or attribute of the integrated view has been modified. If so, the corresponding high-level event is generated to which the list of relevant changes is associated as parameter.

- High-level events related to extensional changes.

  In this case, the same *Modification Manager* processor used in the previous case generates high-level events corresponding to extensional changes. Three types of high-level events can be generated: *Extension modified on table_name T, Extension affected on class_name C, Extension affected on an attribute A of a class C.* For the first case, high-level events are generated for the detected extensional changes occurred in tables such that they have defined database dependencies used by the Translator component. For the second and third cases, high-level events are generated for the detected extensional changes that affect a class or an attribute of the integrated view. In order to achieve this, the *Modification Manager* has to make queries in the component databases. These queries are constructed by looking up the changes stored in the incremental table by the triggers defined over the tables and the mapping informations for the classes and attributes of the integrated view.

Supposing that this is in the incremental table:

<5,J.Smith> inserted in PERSON(ID,NAME)

and a class SUPER_PARENT exists with mapping[8]

$<\mathit{id},\sigma_{\mathrm{new}\geq 3}\ ((_{\mathrm{idp}}\,\mathcal{F}_{\mathrm{count(idc)}}\ (\mathit{has\_child} \bowtie_{\mathrm{(idp=id)}} \mathit{person})))>$

The Modification Manager will know that the extension of the class SUPER_PARENT has been affected (the person with id 5 is an instance of it, but not necessarily a new instance) if the next query gives a non-empty answer:

```
select * from HAS_CHILD, PERSON
where ID=5 AND ID=IDP
group by ID
having count(IDC) >= 3
```

Another important decision is *when to generate* high-level events. Not surprisingly, identifying relevant changes is an expensive operation, it cannot be executed every time a change happens in a database because the component and autonomous databases cannot be overloaded with tasks related to the FDBS. Therefore, the Modification Manager, with a certain time granularity, looks for relevant intensional changes or extensional changes in tables that form part of

[7] The System Consistency Catalog contains the relevant catalog information before the last change for all the databases residing on the same node.

[8] Our mapping information is expressed in terms of the extended relational algebra.

the mapping information of classes and attributes and generates the corresponding high-level events. In general, the time granularity will be decided by the PRI although a default one is provided. Nevertheless, this type of behaviour could admit, during a period of time, some inconsistencies among component and integrated view and some inconsistencies among data in the cache memory and in the component databases.

In figure 2 the enlarged architecture of our FDBS with the processors Modification Detector, Modification Manager and Consistency Re-establisher[9] that form the new component of the system called the Monitor is shown. Detailed features about the different processors of the Monitor component appear in [IBM$^+$95].

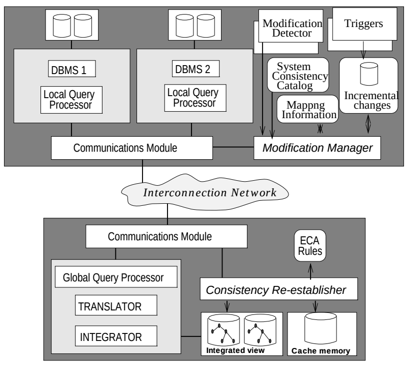

Figure 2: Architecture of the FDBS with the Monitor component.

3 Conditions: Types and treatment

We present the types of conditions upon parameters associated to each type of event[10] (see figure 3). Conditions are verified as soon as a rule is fired by an event. Depending on the type of change, intensional or extensional, the complexity of verifying them is different.

A. Conditions for intensional changes

The conditions corresponding to changes in the schemata are very simple to verify because a list of relevant changes is associated as parameter with each event. For example:

Schema modified on schema_products (DROP_TABLE PRODUCT)

Notice that the enumerated changes correspond to effects observed in the database catalog and not to the real operations.

B. Conditions for extensional changes in tables

Conditions corresponding to extensional changes are much more expensive to check because *they are in general queries to databases.* When the query gets an empty answer then the condition is false, and if the answer is not empty then the condition is true.

For example, if some extensional change has occurred in table T then it is possible that an inclusion dependency of the form $T.Key \subseteq R.Key$ has been violated. The condition $not(T.Key \subseteq R.Key)$ is a case in point. To check this condition implies to ask the next SQL query:

```
select Key from T minus select Key from R
```

When the answer is empty then the condition is not verified but if not, there is some value in T.Key that does not appear in R.Key and therefore, the inclusion dependency is violated and

[9]The Consistency Re-establisher is the processor that manages the Event-Condition-Action rules. Its goal is to re-establish the consistency of the integrated view and cache memory.

[10]Afterwards, with the word event we are referring to high-level events.

Event type	**Condition**
Schema modified on DB_name **Par:** list of changes	Addition/Deletion of table T Addition/Deletion of attribute A in table T Modification of attribute A in table T Addition/Deletion of integrity constraint I
Extension modified on table T **Par:** list of changes	Inclusion dependency involving T violated Exclusion dependency involving T violated Functional dependency involving T violated Attribute of T does not take any NULL value Domain values of an attribute violated
Extension affected on class C **Par:** list of changes	Equivalence of classes involving C violated Inclusion of classes involving C violated Overlapping of classes involving C violated Disjoinment of classes involving C violated Class C is cached
Extension modified on attribute A **Par:** list of changes	Equivalence of attributes involving A violated Inclusion of attributes involving A violated Overlapping of attributes involving A violated Disjoinment of attributes involving A violated Attribute A is cached for C

Figure 3: Event types and their corresponding conditions

the condition verified. However, notice that there are some situations where the execution of the previous SQL query can be avoided: a) when there have been only *deletions* on T but no *insertions* then the condition *not(T.Key* $\subseteq$ *R.Key)* is false; b) when there have been *modifications* on attributes different than *Key* then condition *not(T.Key* $\subseteq$ *R.Key)* is false.

Moreover, the previous SQL query can be transformed into a more efficient one (scanning only R instead T and R) when the number of new values inserted or modified on *Key* is only one (e.g. value k).

```
select * from R where Key=k
```

In the last case, if the answer is empty then the condition is true and false otherwise.

According to the previous situations it can be observed that, *the verification of the condition for the same ECA rule is made in a different way* (avoiding the construction of a SQL query or using a distinct SQL expression) *depending on the particular values of the event parameters.*

In [BW93] a method for improving the efficiency of condition evaluation is presented. They try to verify the conditions only with the incremental changes[11] whenever is possible. This method is applicable to our context. However, there exist some situations in our case that can be optimized further. For example, the condition that verifies the consistency of a functional dependency $T.A \rightarrow T.B$ would be: (expressed in SQL instead of extended relational algebra as they do)

```
select * from T
group by A
having count(B) > 1
```

Using their method this condition cannot be optimized because it makes use of aggregate functions. However, this is not the best way to verify if a functional dependency is violated when the incremental changes are known. If i new tuples have been added to T with values $a_1, \ldots, a_n$ for the attribute A and $b_1, \ldots, b_n$ for the attribute B then the functional dependency is violated if the next query gives a non-empty answer:

```
select * from T
where (A=a1 and B<>b1) or ... (A=aN and B<>bN)
```

Finally, when the DBMS with which the component database has been defined permits the definition of integrity constraints and guarantees that they are satisfied at any time, then the corresponding ECA rules do not have to be defined.

[11]The incremental changes are stored in *delta relations.*

C. Conditions for extensional changes in classes

There are two kinds of conditions related to extensional changes that affect classes: about extensional properties and about cached classes and attributes.

C1. Conditions about extensional properties

The SQL queries corresponding to conditions that check extensional properties between classes[12] are even more expensive than the SQL queries corresponding to extensional changes in tables, because they are usually multidatabase queries. For example, if some extensional change that affects class C has occurred then, it is possible that the correspondence that expresses that C and D are disjoint classes, represented as $RWS^{13}(C) \cap RWS(D) = \emptyset$, has been violated. The condition $not(RWS(C) \cap RWS(D) = \emptyset)$ is a case in point. Checking this condition implies to execute the following multidatabase SQL query:

```
select id_C from rel_for_C
  intersect
select id_D from rel_for_D
```

If a multidatabase SQL is available then the query can be directly made. If not, different SQL queries need to be sent to each database and the final answer built. Notice also that `rel_for_C` and `rel_for_D` may also be derived relations obtained from different databases.

Notice that it is not needed to execute that SQL query in the following cases: when there have been only *deletions* on C but not *insertions* then condition $not(RWS(C) \cap RWS(D) = \emptyset)$ is false; when there have been *modifications* on attributes different than *id_C* then condition $not(RWS(C) \cap RWS(D) = \emptyset)$ is false.

Moreover, the previous SQL query can be transformed into a more efficient one (with only one table scan instead of two) when the new values inserted or modified on *id_C* are known *(c1,...,cN)*.

```
select * from rel_for_C
where id_C=c1 or ... or id_C=cN
```

In this last case, if the answer is empty then the condition is true and false otherwise.

C2. Conditions for cached classes/attributes

Conditions corresponding to ECA rules that ensure the consistency of the cache memory are very simple to verify because it is only required to search in a list of cached classes and attributes (a list with the names of all the cached classes and attributes already exists). In fact, the hardest task is to know when the extension of a class has been affected by a modification because in the component databases there are tuples of tables and not instances. This task is achieved by the Modification Manager.

4 Actions

The action part of an ECA rule should compound the inconsistencies created by extensional or intensional changes. The main point here is whether the consistency can be automatically recovered or not. If not, at least some warning must be sent to the PRI and the users informing them about the nature of the detected change. The different actions that can be invoked in ECA rules are:

- Pre-defined operations that permit the automatic re-establishment of the consistency of the integrated view. This set of pre-defined operations includes the set used during the construction of the integrated view:

[12] We do not discuss about extensional changes that affect attributes because they are treated in an analogous way.

[13] Real World State (RWS), that is the real world counterpart of elements in the view.

OPERATION	**parameters**
ADD-CLASS	*name,description,map-info-class*
ADD-ATTRIBUTE	*name,description,map-info-attr*
GENERALIZE-CLASS	*name,list of class-names,description*
GENERALIZE-ATTRIBUTE	*name,list of attr-names,description*
SPECIALIZE-CLASS	*class-name,description*
SPECIALIZE-ATTRIBUTE	*attr-name,description*
DELETE-CLASS	*class-name*
DELETE-ATTRIBUTE	*attr-name*
REDEFINE-CLASS	*class-name,description*
REDEFINE-ATTRIBUTE	*attr-name, description*
MODIFY-MAPPING-INFO-CLASS	*class-name,map-info-class*
MODIFY-MAPPING-INFO-ATTR	*attr-name,map-info-attr*

There are also other actions to re-establish the cache memory:

OPERACION	**parameters**
INSERT-INSTANCES	*class-name,list-of-instances*
DELETE-INSTANCES	*class-name,list-of-instances*
INSERT-ATTRIBUTE-VALUES	*attr-name,list-of-attr-values*
DELETE-ATTRIBUTE-VALUES	*attr-name,list-of-attr-values*

- Warnings to the users of the integrated view. Warnings are generated in the case that automatic actions cannot be made. These warnings will appear every time a user starts a session with the inconsistent integrated view. It is the PRI's responsibility to eliminate them.

- Calls to specific procedures defined by the PRI. These specific procedures allow the PRI to customize the behavior of the system. For example, although in general nothing can be automatically done when a new table is added, the PRI may know what to do for a particular table created in a database and define whatever actions he wants.

All the previous actions are valid only when the rule has been successfully executed. But if a rule aborts during its execution then the changes produced by the actions have to be re-established. In figure 4, some examples of different types of actions in ECA rules are presented.

A	**when** *Schema modified on schema_products* (DROP_TABLE PRODUCT) **if** *deleted table PRODUCT* **do** *DELETE class PRODUCT from the integrated view*
B	**when** *Extension modified on table_name TEACHES* (ADDED_VALUES TEACHES <17,"S43">) **if** *not(TEACHES.subject* $\subseteq$ *SUBJECT.name)* **do** *WARNING('Call to the PRI')*
C1	**when** *Extension affected on class_name TEACHING_ASSISTANT* (DELETED_VALUES TEACHING_ASSISTANT <18>,<19>) **if** *not(RWS(TEACHER)* $\cap$ *RWS(STUDENT)* $\neq \emptyset$*)* **do** *DELETE class TEACHING_ASSISTANT from the integrated view*
C2	**when** *Extension affected on class_name TEACHING_ASSISTANT* (DELETED_VALUES TEACHING_ASSISTANT <18>,<19>) **if** *TEACHING_ASSISTANT is cached* **do** *DELETE (<18>,<19>) on TEACHING_ASSISTANT*

Figure 4: Examples of some ECA rules, instantiated with the parameters.

5 Conclusions

In this paper we have presented the inclusion of an active mechanism into a Federated Database System. The goal of this mechanism is to maintain the consistency of the integrated view and the cache memory with respect to the component databases. The used knowledge model for the mechanism has been the Event-Condition-Action rules. We have distinguished between two levels of events: low-level events (related to intensional and extensional changes on the component databases) and high-level events (related only to relevant changes). Only the last type of events are expressed in the ECA rules. How those events are detected and generated has been explained. We have also shown the types of conditions and how they can be verified. In some cases the complexity of verifying them is big. It is worth mentioning that the verification of some conditions can be made in a different way depending on the particular values of the event parameters. And finally, we have pointed out the different types of actions that try to re-establish the consistency of the integrated view.

References

[ASD+91] R. Ahmed, P. Smedt, W. Du, W. Kent, M. Ketabchi, and W.A. Litwin. The Pegasus heterogeneous multidatabase system. *IEEE Computer*, 24:19–27, December 1991.

[BBMR89] A. Borgida, R.J. Brachman, D.L. McGuinness, and L.A. Resnick. CLASSIC: A structural data model for objects. In *Proceedings ACM SIGMOD-89, Portland, Oregon*, 1989.

[BG92] D. Bell and J. Grimson. *Distributed Database Systems.* Addison-Wesley, 1992.

[BIG94] J.M. Blanco, A. Illarramendi, and A. Goñi. Building a federated database system: an approach using a knowledge based system. *International Journal on Intelligent and Cooperative Information Systems*, 3(4):415–455, December 1994.

[BW93] E. Baralis and J. Widom. A rewriting technique for using delta relations to improve condition evaluation in active databases. Technical report, Department of Computer Science. Stanford University. CS-93-1495, November 1993.

[CHS91] C. Collet, M. N. Huhns, and W. Shen. Resource integration using a large knowledge base in CARNOT. *IEEE Computer*, pages 55–62, December 1991.

[CT91] B. Czejdo and M. Taylor. Integration of database systems using an object-oriented approach. In *First International Workshop on Interoperability in Multidatabase Systems*, April 1991.

[CW93] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proc. of the VLDB*, 1993.

[DBM88] 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.

[GD93] 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.

[GIMB97] A. Goñi, A. Illarramendi, E. Mena, and J.M. Blanco. An optimal cache for a federated database system. To appear in *Journal of Intelligent Information Systems.*, 1997.

[IBM+95] A. Illarramendi, J. M. Blanco, E. Mena, A. Goñi, and J. M. Pérez. Maintaining schemata consistency for interoperable database systems. In *Proceedings of the Fourth International Conference on Database Systems for Advanced Applications (DASFAA'95).* World Scientific Publishing, April 1995.

[LNE89] J. A. Larson, S. B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE TOSE*, SE-15(4), April 1989.

[PSKQ89] C. Peltason, A. Schmiedel, C. Kindermann, and J. Quantz. The BACK system revisited. Technical University Berlin, September 1989.

[RSK91] M. Rusinkiewicz, A. Sheth, and G. Karabatis. Specifying interdatabase dependencies in a multidatabase environment. *Computer. December*, 24(12), 1991.

[SGN93] A.P. Sheth, S.K. Gala, and S.B. Navathe. On automatic reasoning for schema integration. *International Journal on Intelligent and Cooperative Information Systems*, 2(1):23–50, 1993.

[SK92] 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):185–191, April 1992.

[SK93] L.J. Seligman and L. Kerschberg. An active database approach to consistency management in data- and knowledge-based systems. *International Journal on Intelligent and Cooperative Information Systems*, 2(2), 1993.

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB*, 1:81–126, 1992.