Implementing Mediators through Virtual Updateable Views[1]

Hanna Kozankiewicz[*], Jacek Leszczyłowski[*], Kazimierz Subieta[*#]

[*]) Institute of Computer Science, Polish Ac.Sci., Warsaw, Poland
[#]) Polish-Japanese Institute of Information Technologies, Warsaw, Poland
{hanka, jacek, subieta}@ipipan.waw.pl

Abstract. Mediators are considered basic architectural units for integration of distributed, heterogeneous information resources. In the paper we propose powerful virtual updateable views as a very high-level tool for their implementation. We assume that a view definer can explicitly determine view updates intention through procedures (being a part of a view definition) which dynamically overload generic view updating operations. We follow the Stack-Based Approach, a new theory of object-oriented query languages based on the classical concepts of programming languages, such as the environment stack and the naming-scoping-binding paradigm. The proposal addresses very general object-oriented model and offers full computational power of a mediator definition language.

1 Introduction

The growth of business-oriented applications of the Internet has caused the necessity to provide mechanisms for integration of distributed and heterogeneous data/service resources. The topic has already been investigated in various contexts, such as federated databases and CORBA-oriented applications. A new term - mediator [Wied92] - has been coined to denote an architectural unit of a such integration. Mediators transform data from one format into another and provide generalization and abstraction over data. They also combine, refine, and integrate data from multiple sources.

An unsolved issue concerning mediators concerns the methodology, tools and rules that can be efficiently used for their design and construction. Till now, this part of the technology seems to be immature. Typically, mediators are implemented in lower level programming languages like C++ or Java. Such an approach has disadvantages - an increased time and cost of development, clumsy and long code, poor reuse, high maintenance cost and time. There are proposals to implement mediators in declarative languages [BRU96]. Following them we claim that mediator functionalities can be provided by means of powerful database virtual views. The use of views to integrate heterogeneous ontologies has already been discussed in [Subi01].

Apart from advantages of database views as tools for implementing mediators there are several difficult problems. We identify them as follows:

---

[1] Supported by the European Commission 5th Framework project ICONS (Intelligent Content Management System); no. IST-2001-32429.

- Computational/algorithmic universality of the view definition language. Standard SQL is not sufficiently universal, hence in many cases it cannot be used to specify mediators.
- Data model and data structures that view definitions address. SQL addresses only relational structures, but majority of Web resources are or will be based on XML-oriented data, or even more complex RDF-oriented, object-oriented or object-relational structures.
- The view updating problem. This concerns mediators that have to update source databases. Current solutions of the problem are limited, despite of intensive research. Limited view updating possibilities reduce applicability of the idea.
- Performance. A mediator should not imply significant performance penalty. This means new needs for optimization of queries involving views.

Views are the subject of many research efforts in the context of XML technologies [Abit00], object-oriented and object-relational databases [ABS97, LaSc91, SLT91]. There are some implemented prototypes like views in O2 [Souz95], ActiveViews [AAC+99], or stored functional procedures in Loqis [Subi91]. View concepts occur also in the ODMG standard [ODMG00] (the "define" statement of OQL) and in the SQL:99 standard [MSG01]. Nevertheless, in our opinion no current proposal concerning views presents satisfactory idea how to cope with the above mentioned problems. This is the motivation of our research, which addresses all of them.

Our idea is simple and straightforward. A view definition is a complex module that consists of not only a single query (as in SQL) but contains a part that allows the view definer to take full control over view updates. The part consists of procedures that dynamically overload generic view updates of virtual objects. The procedures are defined on top of a query language [SKL95]. The view definer should write an appropriate procedure for each of necessary view updating operations. Queries involving such views are optimizable through the query modification technique [SuPl01]. The idea is exactly in the spirit of the *instead of* triggers of Oracle and MS SQL Server, but it is much more general and applicable to non-relational database/web systems.

The approach has obvious consequences:

- Query language must be computationally complete and must address a complex object-oriented model (covering other models - relational, XML, etc.)
- On top of the query language there must be defined imperative statements a la SQL *update*, *insert* and *delete*. Such statements can be involved into control statements such as *if*, *while*, *for each*, etc.
- On top of the above there must be defined functions and procedures, with parameters, a la SQL stored procedures (in the style of Oracle PL/SQL).

The above assumptions exclude the traditional approaches to query languages based on relational algebras, calculi, logic and their object-oriented counterparts. The view mechanism presented in this paper is defined within the Stack Based Approach (SBA) to query languages [SKL95]. The approach has roots in the semantics of programming languages. It integrates query languages' and programming languages' concepts into a unified, consistent and non-redundant system of notions. It is abstract and universal, which makes it relevant to a general object model. The approach has already been implemented in a prototype Loqis [Subi91] (an object-oriented DBMS), in a prototype addressing an object model with dynamic object roles [SJHP03], in a prototype of an XML-oriented query language for the DOM model [HP02], and in

other prototypes. Currently we are working on a prototype for the EU project ICONS, where all features presented in this paper have to be implemented.

The presented approach meets the requirements included in the report of I^3/POB working group [BRU96] that is devoted to a standard language for mediators: it supports complex data types and for their arbitrary combination, semi-structured data, abstract data types (as a particular case of the class concept), and other required features.

The rest of this paper is structured as follows. In the next section we formalize data store structures. Section 3 introduces the notion of an environment stack and describes its main roles i.e. name scoping and binding. Section 4 presents the Stack Based Query Language (SBQL). Section 5 introduces our approach to updateable views and Section 6 presents examples illustrating its power. Section 7 concludes.

2 Abstract Object Store Model

In order to formalize a language for view/mediator specification we have to formalize data structures that it addresses. In the paper we focus on simple data stores, which contain nested objects and relationships between them. We would like to emphasize that in SBA the store model and its query language can be easily extended to very complex object models, including notions of classes, inheritance, and dynamic roles (see [SJHP03]).

In SBA each object consists of the following components:

- Internal identifier (OID) that is automatically assigned and identifies the object.
- External name (introduced by a designer, programmer or user) used to access the object from a query or program.
- Content that can be a value, a link, or a set of objects.

Let I be a set of the internal identifiers, N be a set of the external names, and V be a set of the atomic values, e.g. numbers, strings, blobs, etc. Atomic values include also codes of procedures, functions, methods, views, and other procedural entities. Formally objects are modeled as triples defined below, where i, i_1, $i_2 \in I$, $n \in N$, and $v \in V$:

- Atomic objects *as* $<i, n, v>$.
- Link objects *as* $<i_1, n, i_2>$. that model relationships between objects.
- Complex objects *as* $<i, n, S>$, where S denotes a set of objects.

Note that this definition is recursive and allows one to create linked and compound objects with an arbitrary number of hierarchy levels. To model collections SBA does not impose uniqueness of external names at any level of data hierarchy (as in XML).
In SBA, objects populate an object store, which consists of the following elements:

- The structure of objects, subobjects, etc.
- OIDs of root objects, which are accessible from the outside (starting points for querying).
- Constraints (e.g. uniqueness of OIDs, referential integrities, etc).

Example data store. A schema of the store presented in Fig.1 describes a part of a database for a bookstore. Fig.1 also contains a sample state of the object store.

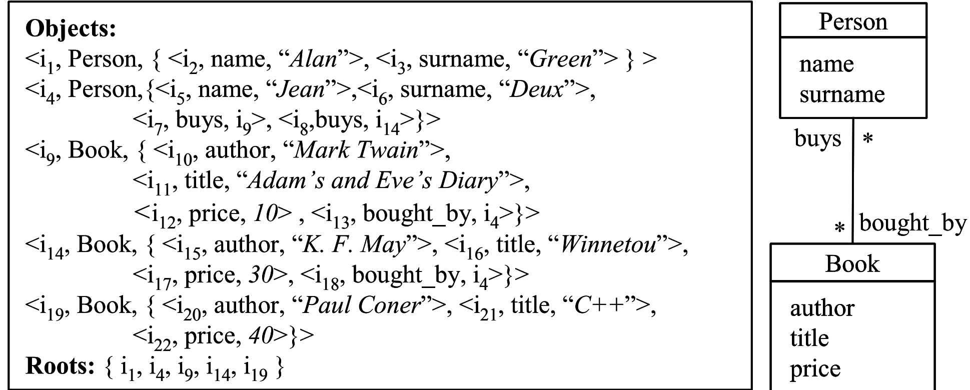

Fig. 1. The example object store and its schema

3 Name Binding

The foundation of SBA is the naming-scoping-binding mechanism, which ensures that each name occurring in a query is bound to an appropriate run-time entity (an object, attribute, method, parameter, etc.) according to the scope of this name. Scope control and name binding is managed using an environment stack (ES) – a structure well-known in programming languages. The stack enables the *abstraction principle*, which allows the programmer to consider the currently being written piece of code to be independent of the contexts of its possible uses.

ES consists of *sections* that contain sets of *binders*. A binder is an SBA concept used to cope with various naming issues that occur in object models and their query languages. Formally, a binder is a pair (n, x), where n is an external name, and x is a reference to an object; such a pair is written as $n(x)$. We refer to n as the *binder name*, and to x as its *binder value*. The concept of a binder is generalized; in particular, x can be an atomic value or a compound structure.

In SBA, at the beginning of a user session ES consists of base sections containing binders for all root database objects. Usually there exist other base sections with binders to computer environment variables, to local objects of the user session, to libraries, etc. During query evaluation the stack grows and shrinks according to query nesting. Assuming there are no side effects in queries (no calls of updating methods), the final ES state is exactly the same as the initial one.

The process which determines the meaning of an external name is called *binding*. Binding follows the "search from the top" rule: to bind a name n the binding mechanism is looking for the ES "visible" section closest to the top of the stack and containing a binder having the considered name n. If the binder is $n(x)$ then the result of the binding is x. To cover collections, SBA allows that the binding can be multi-valued, that is, if the relevant section contains several binders whose names are n: $n(x_1)$, $n(x_2)$, $n(x_3)$,..., then all of them contribute to the result of the binding. In such a case the binding of n returns the collection $\{x_1, x_2, x_3, ...\}$. Fig.2 presents an example ES with base section containing binders to root objects of the store showed in Fig.1 and with

one additional section at the top containing binders representing the content of the object i_4. Arrows indicate the order of name binding search.

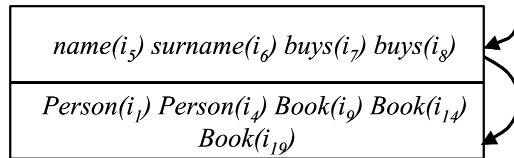

Fig. 2. Name binding

Function nested. Function nested allows constructing new section(s) on the top of ES. For an element r the function returns a set of binders in the following way:

- If r is a single identifier of a complex object, e.g. i_4, then *nested* returns binders to subobjects of given object (in Fig.2 the top section of the stack contains the internal environment of *Person*(i_4)).
- If r is an identifier of a link object, e.g., i_7, then *nested* returns binders to the object the link points to (e.g. for i_7 it is {*Book*(i_9)}).
- If r is a binder, then *nested* returns {r} (a set consisting of the binder).
- If r is a structure *struct*{r_1, r_2, r_3, ...}, then *nested* returns the sum of results returned by *nested* for r_1, r_2, r_3, ...
- For other r *nested* returns the empty set.

4 Stack Based Query Language (SBQL)

SBQL [SKL95] is a formalized query language in the spirit of OQL. Its syntax is:

- A single name or a single literal is an (atomic) query. For instance, *Person*, *name*, *y*, “Smith”, 2500, etc, are queries. Note that the identifiers are not representable.
- If q is a query, and σ is a unary operator (e.g. sum, count, distinct, sin, sqrt), then $\sigma(q)$ is a query.
- If q_1 and q_2 are queries, and θ is a binary operator (e.g. *where*, dot, *join*, =, *and*), then $q_1 \theta q_2$ is a query.

SBQL is based on the principle of operator orthogonality, which means that operators can be freely combined unless it violates some type constraints. We distinguish algebraic and non-algebraic operators – the subdivision is explained below.

Query Results. We denote the set of results of SBQL queries *Result* and elements of this set *q_values*. SBQL queries can only return one of the following *q_value*:

- atomic value (e.g. 3, ‘Smith’, TRUE, etc.)
- reference to an object (of any kind, e.g. i_1, i_5, i_8, i_{18}.)
- binder n(v), where v is a *q_value* and n is any name.
- structure *struct*{ v_1, v_2, v_3, ...} where v_1, v_2, v_3, ... are *q_values* and *struct* is a structure constructor. In general, an order of elements is essential. This constraint can be relaxed providing all v_i are binders. This construct generalizes a tuple known from the relational model.

- collections: *bag*{ v_1, v_2, v_3,,... }, *sequence* { v_1, v_2, v_3, ... }, … where *bag*, *sequence*, ... are collection constructors, and v_1, v_2, v_3, ... are q_values.

Algebraic operators. The operator is algebraic if its semantics is expressed without involving ES. Algebraic operators include numerical, string and boolean comparisons and operators, aggregate functions, set, bag and sequence comparisons and operators , the Cartesian product, etc. Let q_1 and q_2 be queries and Δ be a symbol denoting a binary algebraic operator Δ. Then, in order to evaluate the query $q_1\ \Delta\ q_2$, queries q_1 and q_2 are evaluated independently and then Δ is performed on two partial results (taking them in the proper order), returning the final result.

Non-algebraic operators. Non-algebraic operators are defined in terms of ES. They include selection, projection/navigation, dependent join, quantifiers, and others. We have rejected the common view that these operators can be "algebralized" in the relational algebra style. This can be done only by shifting a part of the semantics to the informal meta-language of mathematics, and we would like definitely to avoid that. If the query $q_1\ \theta\ q_2$ involves a non-algebraic operator θ, then q_2 is evaluated in the context determined by q_1. Thus, the order of evaluation of sub-queries q_1 and q_2 is significant. These operators are called "non-algebraic" because they do not follow the basic property of algebraic expressions, i.e. independent evaluation of q_1 and q_2.

The query $q_1\ \theta\ q_2$ is evaluated as follows. For each element *r* of *q_value* returned by q_1, the subquery q_2 is evaluated. Before such an evaluation, a new section *nested*(*r*) is pushed on ES. After the evaluation ES returns to the previous state. A partial result of the evaluation is a combination of *r* and the *q_value* returned by q_2 for that *r*; the combination depends on θ. Partial results are merged into the final result.

Procedures. SBQL incorporates procedures, with or without parameters, returning an output or not. A procedure parameter can be any query. We adopt *call-by-value*, *call-by-reference* and other parameter passing methods. There are no limitations on computational complexity, what can be useful for view definitions when the mapping between stored and imaginary objects is complex and requires non-trivial algorithms. The results of functional procedures (functions) belong to the same semantic category as results of queries, therefore they can be invoked in queries. Due to ES, there are no restrictions on calling functions within the body of (other) functions, what enables among others recursive calls. So far SBQL and the procedures are untyped, but another research group intends to introduce static type checking.

Below, we present an example function *bestsellers* returning books which were sold in more then *nb_sold* pieces; *nb_sold* is a *call-by-value* function parameter:

```
function bestsellers ( in nb_sold ) {
  return Book where count ( bought_by ) > nb_sold;  }
```

A call of this function is shown in a query:

```
Get authors and titles of books that were sold in more then 500 pieces:
        bestsellers( 500 ) . (title, author)
```

5 Updateable Views

In majority of the classical approaches (relational, object-relational and object-oriented) a database view is essentially a functional procedure. View updates are performed through side effects of view definitions, usually through various kinds of references to stored data returned by view invocations. The art of view updating focuses on forbidding updates that may violate user intention, c.f. view updateability criteria, such as no view over-updating.

We abandon this approach and disallow any updates through side effects. Instead, we explicitly introduce information on intents of view updates in the form of procedures that overload generic view updating operations. Our approach to updateable views is described in detail in [KLPS02]. We propose a two-query paradigm to operations on views. The first query preserves all the necessary information about the stored source objects involved in the view, while the second query takes the result of the first query as an input and delivers the final mapping between stored and virtual objects. The first part in the two-query paradigm is a *sack definition*. A sack contains *seeds*, which unambiguously identify stored objects, which "induce" the virtual ones. A seed is a parameter of updating procedures defined by the view definer for determining view updates. This parameter is passed implicitly (it is internal to the proposed mechanism) and the view definer does not need to bother about it.

We distinguished the following operations on virtual objects:

- deletion of a given object.
- insertion of a new object inside a given object (inserted object's id is a parameter).
- dereferencing that returns the value of an object.
- updating the value of a given object. The operation has a new value as a parameter.

A view definer has to write an overloading procedure for each operation that he/she wish to support. If some overloading procedure is not defined, then the corresponding operation is not supported (it is forbidden). We ascribed fixed procedure names (*on_delete*, *on_insert*, *on_retrieve*, *on_update*) to these operations, which have a different syntactic representation in a programming language. Names of formal parameters of procedures *on_insert* and *on_update* can be chosen by the view definer. We provide a typical method of passing parameters to procedures bodies through binders inserted into a corresponding ES activation record.

View definition can also include definition of sub-views. Our idea is based on the *relativity principle*, which assumes that each nested entity has the same syntactic and semantic properties as the external one. Hence sub-views are defined in the same way.

We can define a view *ClientDef* with information about persons who have ever bought a book in our bookstore. We define operation of dereference returning client name and we allow deletion of the client. The view definition may looks as follows:

```
create view ClientDef {
    virtual objects Client {(Person where count( buys ) > 0) as c;}
    on_retrieve do { return c . name; }
    on_delete do { delete c; }}
```

We distinguish in a uniform way the view definition name from the name of virtual objects generated by the view. In the example we use names *ClientDef* (required for view creating, updating, and deleting) and *Client* (that identifies virtual objects).

View definitions are kept in an object store. An ES database section has to contain both: a binder to the view definition (required to use/change the view's definition) and a binder to the sack definition (that allows querying/updating virtual objects). Observe, that this allows the separation of a view definition from existence of virtual objects. Thus it is possible to have a view defined in some place and not "activated" – with no virtual objects existing. Besides, one may have virtual objects derived according to the same view definition several times and in each of the times they may exist in a different place.

View Call Processing. View call processing requires passing of information on virtual objects to the proper updating procedure defined within the view. The system has to determine whether it deals with a stored or virtual object, which view, and which virtual object. Therefore, we have introduced a *virtual identifier* of the form:

<*Flag "I am virtual", View definition identifier, Seed*>

When a system processes a virtual identifier it pushes on ES a section containing *nested*(*seed*) and a section with binders to all sub-views definitions of the processed view. In such a way we pass the *seed* parameter to all the (dereferencing, updating) procedures that and make all subviews (i.e. attributes of virtual objects) available for querying.

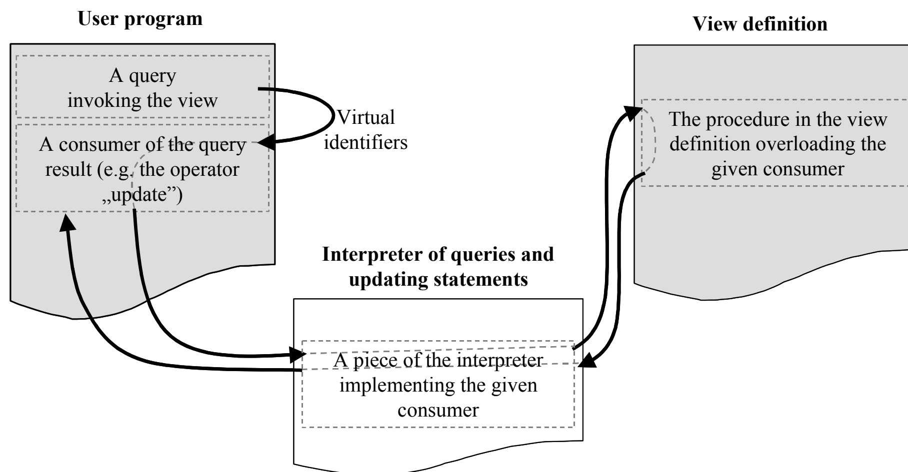

Fig. 3. Processing of a view call.

In Fig.3 we depicted a flow of control during processing a view update operation. First, the query involving a view is evaluated and it returns virtual identifier(s). A user program determines which operation should be performed and passes the control to the query interpreter. It recognizes that the operation concerns a virtual object and passes control to a proper procedure from the proper view definition. When the procedure execution is finished, the control returns to the user program.

Views with Parameters and Recursive Views. Views, similarly to procedures, can have parameters. A parameter can be any query - in particular, it may return a bag $bag(r_1, r_2, ..., r_n)$ where $r_1, r_2, ..., r_n \in$ *Result*. A query interpreter determines a method of parameters passing basing on the view definition syntax. We are going to implement a method known as *strict-call-by-value*, in which the result of a query being a parameter is passed without any change to the procedure's body. Technically, it means that if for a formal parameter *par* the query returns a result *r*, then to the corresponding activation record for procedure *p* is added a single binder *par*(*r*). In this way the result of the query becomes available within the *p* body under the name *par*. Currently, in our approach parameters concern only seed definitions and are not available for definitions of updating procedures (it is possible, but requires small extensions of the mechanism).

Recursion is given for free within the stack-based semantics. As shown, views are programming entities like functions and procedures. All volatile data created by the view are pushed on ES, thus each view call is independent on other view calls. Hence, recursive views are fully supported by the described mechanism

Nested views. For nesting views we have to extend the notion of virtual identifiers for virtual objects from subviews to enable access to all seeds of parent virtual objects along the path of nesting. A possible extended form of a virtual identifier is:

<*Flag "I am virtual", (View definition identifier*$_1$*, Seed*$_1$*), ..., (View definition identifier*$_n$*, Seed*$_n$*)* >

where (*View definition identifier*$_1$*, Seed*$_1$) refers to the most outer view, and (*View definition identifier*$_n$*, Seed*$_n$) refers to the current inner view.

When an identifier is processed by any of the procedures *on_retrieve*, *on_update*, *on_delete*, or *on_insert*, the interpreter pushes on ES a section containing *nested*(*Seed*$_1$)$\cup$ *nested*(*Seed*$_2$)$\cup$ ... $\cup$ *nested*(*Seed*$_n$), and then calls the proper procedure. This is the way of passing information on seeds to all the procedures.

6 View Examples

In this section we present examples illustrating power of the approach. In the examples we use the database presented in Fig.1.

Currency conversion. We define a view that for each book returns a virtual object containing book's title and its price in Euro (we assume that prices are kept in the database in USD). The view defines an operation of a price rise, where an amount of rise is given in Euro (the corresponding procedure transforms it in USD and raises the price). The view definition looks as follows:

```
create view BookTitleEuroPriceDef {
  virtual objects BookTitleEuroPrice{ return Book as b; }
  create view BookTitleDef {
    virtual objects BookTitle { return b.title as bt; }
```

```
    on_retrieve do { return bt; }    }
  create view EuroPriceDef {
    virtual objects EuroPrice { return b.price as bp; }
    on_retrieve do { return bp * CurrentDollarToEuroExchangeRate(); }
    on_update  new_euro_price do {
      if new_euro_price < 0 then { print("Error: New book price < 0?"); return;}
      else bp := new_euro_price /CurrentDollarToEuroExchangeRate();  }  }  }
```

Call of the view in a query that decreases the price of the "Winnetou" book on 10 Euro:

```
for each BookTitleEuroPrice where BookTitle = "Winnetou"  do
                EuroPrice:= EuroPrice - 10;
```

Providing security. We define a view returning information on clients and books they have bought in a bookstore. The information about books is public, unlike information about clients that should be available only for authorized users. The view should protect these data against hackers by returning false results instead of forbidding access. The example is under influence of [SJGP90].

```
create view BooksClientInfoDef {
  virtual objects BooksClientInfo { return Book as b;}
  on_retrieve do { return b; }

  create view ClientDef {
    virtual objects Client { return b.bought_by.Person.surname as c;}
    on_retrieve do {
      if AccessIsAuthorised() then return c;
      else {
        create local FalseClients :=  sequence {"Smith", "White", "Black"};
        return selectRandomCombinationOf( FalseClients );        } } } }
```

7 Summary

We have presented an approach to implementing mediators through very powerful object-oriented updateable views. The mechanism is based on the Stack Based Approach to query languages. We have shown that the presented approach allows one to define very powerful views in which the view definer has full control over what happens with updates of virtual objects. We provide using such views to integrate heterogeneous data resources in federated database and/or web systems.

Our future work on the presented view mechanism, after finishing the implementation for an XML-oriented store, will focus on extending the concept on more complex data stores and on introducing new extensions to the view mechanism e.g. introducing stateful views, classes for virtual objects, etc. which could be very useful in the context of mediators.

References

[AAC+99] S.Abiteboul, B.Amman, S.Cluet, A.Eyal, L.Mignet, T.Milo. Active Views for Electronic Commerce. Proc. of VLDB Conf., 1999, 138-149.

[Abit00] S.Abiteboul. On Views and XML. Proc. of PODS Conf., 1999, 1-9

[ABS97] S.Amer-Yahia, P. Breche, and C. Souza dos Santos. Objects Views and Updates. Engineering of Information Systems Journal 5(1), 1997.

[BRU96] P.Buneman, L.Rashid, J.Ullman. Mediator Languages - a Proposal for a Standard. Report of an I3/POB Working Group, University of Maryland, April 1996

[HP02] R.Hryniów, T.Pieciukiewicz. A Stack-Based XML Query Language. Master's thesis, Polish-Japanese Institute of Information Technology, 2002.

[KLPS02] H.Kozankiewicz, J.Leszczyłowski, J.Płodzień, and K.Subieta. Updatable Object Views. Institute of Computer Science, Polish Ac.Sci, Report 950, 2002

[LaSc91] C.Laasch, M.H.Scholl, M.Tresch. Updatable Views in Object-Oriented Databases. Proc. of 2nd DOOD Conf., Springer LNCS 566, 1991

[MSG01] J.Melton, A.R.Simon, J.Gray. SQL:1999 - Understanding Relational Language Components. Morgan Kaufmann Publishers, 2001

[ODMG00] Object Data Management Group: The Object Database Standard ODMG, Release 3.0. R.G.G.Cattel, D.K.Barry, Ed., Morgan Kaufmann, 2000

[SJGP90] M.Stonebraker, A.Jhingran, J.Goh, S.Potamianos: On Rules, Procedures, Caching and Views in Data Base Systems. Proc. of SIGMOD Conf., 1990, 281-290

[SJHP03] K.Subieta, A.Jodłowski, P.Habela, J.Płodzień. Conceptual Modeling of Business Applications with Dynamic Object Roles. (in) “Technologies Supporting Business Solutions”, The ACTP Series, Nova Science Books and Journals, USA, 2003

[SKL95] K.Subieta, Y.Kambayashi, J.Leszczylowski. Procedures in Object-Oriented Query Languages. VLDB 1995: 182-193

[SLT91] M.H.Scholl, C.Laasch, M.Tresch. Updatable Views in Object-Oriented Databases. Proc. 2-nd DOOD Conf. Springer LNCS 566, 1991

[Souz95] C.Souza dos Santos. Design and Implementation of Object-Oriented Views, Proc. of DEXA Conf., Springer LNCS 978, 1995, 91-102

[Subi01] K.Subieta. Mapping Heterogeneous Ontologies Through Object Views. EFIS’01

[Subi91] K.Subieta. LOQIS: The Object-Oriented Database Programming System. Proc. 1st Intl. East/West Database Workshop on Next Generation Information System Technology, Springer LNCS 504, 1991, 403-421

[SuPl01] K.Subieta, J.Płodzień. Object Views and Query Modification, (in) "Databases and Information Systems", Kluwer Academic Publishers, pp. 3-14, 2001

[Wied92] G.Wiederhold. Mediators in the Architecture of Future Information Systems, IEEE Computer Magazine, March 1992