

Updatable XML Views¹

Hanna Kozankiewicz*, Jacek Leszczyłowski*, Kazimierz Subieta^{#*}

^{*}) Institute of Computer Science Polish Academy of Sciences, Warsaw, Poland

[#]) Polish-Japanese Institute of Information Technology, Warsaw, Poland
{hanka, jacek, subieta}@ipipan.waw.pl

Abstract. XML views can be used in Web applications to resolve incompatibilities among heterogeneous XML sources. They allow to reduce the amount of data that a user has to deal with and to customize an XML source. We consider virtual updatable views for a query language addressing an XML native database. The novelty of the presented mechanism is inclusion of information about intents of updates into view definitions. This information takes the form of procedures that overload generic view updating operations. The mechanism requires integration of queries with imperative (procedural) statements and with procedures. This integration is possible within the Stack-Based Approach to query languages, which is based on the classical concepts of programming languages such as the environment stack and the paradigm of naming/scoping/binding. In the paper, we present the view mechanism describing its syntax, semantics and discussing examples illustrating its possible applications.

1 Introduction

XML has become of great interest for both internet and database community as it grows to be one of the most commonly used standards for data representation and exchange on the Web. Development of XML technologies shows a tendency to treat the Web as a semi-structured database consisting of multiple autonomous sites. In this context, views² can be considered as a virtual mapping of heterogeneous resources stored at remote sites to some unified business ontology. Views can offer many features like abstraction and generalization over data, transformation of data, access and merging data from multiple databases, etc. Assuming that views have full computational power, they fulfill the role that is typically assigned to mediators [Wied92], wrappers and adaptors. Such views addressing XML native databases can support indispensable qualities for many Web applications. The following general issues underlined in the database literature have their counterparts in applications of views to XML-based Web resources:

- *Customization, conceptualization and encapsulation.* Since users sharing the same XML data may have different needs, views enable them to see the same data differ-

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

² A view is a virtual image of data stored in a database/XML file. We deal with virtual views. Materialized views are a different research subject, almost entirely irrelevant to this paper.

- ently - tailored to their interests and in the form suitable to their activity.
- *Security, privacy and autonomy.* Views give the possibility to restrict user access to relevant parts of XML data.
 - *Interoperability, heterogeneity, schema integration and legacy applications.* Views enable integration of distributed/heterogeneous data sources, allowing understanding and processing alien legacy or remote databases according to a common, unified schema.
 - *Data independence and schema evolution.* Views enable users to change physical and logical data organization/schema without affecting already written applications.

These qualities are hard to satisfy goals, thus are challenging tasks for research and development. The literature concerning virtual views for object-oriented and object-relational databases has not yet presented a solution that would be satisfactory with respect to semantic clarity, implementability, computational universality, user friendliness and performance. This especially concerns updatable views that imply a difficult problem how to map updates addressing virtual data into updates of stored data.

Recently, views are the subject of research in the context of XML technologies [Abit00, KL02]. There have already been developed prototype view implementations [AAC+98, Lac01]. Papers on implemented views address also the area of semi-structured data [AGM+97, LAW99], closely related to XML data. These contributions show trends, but the subject still requires much more research and development.

In this paper, we present a new approach to virtual, updatable views for a query language addressing XML native databases. Updatability of views requires integration of update operations with query semantics. This excludes the classical approaches to query languages, such as object algebras and various forms of logic/calculi (in particular, an XML query algebra), because these frameworks do not deal with updating. We follow the Stack-Based Approach (SBA) to query languages [SBMS95, SKL95], which expresses the query semantics in terms of classical notions of programming languages, such as the environment stack, naming issues, scopes for names and binding names to run-time database/program entities. Our idea requires integration of queries with procedural statements and with procedures.

The novelty of our approach is introduction of information about intents of view updates into views' definitions. This is a revolutionary change in comparison to known approaches, which as a rule assume updates via some side effects of view definitions (e.g. by OIDs returned by a view invocation). Similarly to "instead of" triggers, the intents have a form of procedures that during run time perform view updating operations. However, in contrast to the "instead of" triggers, our approach addresses non-relational databases, it assumes overloading generic view updating operations by procedures rather than the event-action paradigm, it is much more general and it is optimisable. These features create possibilities, which have not been even considered in other approaches to updatable views, or have been considered as absolutely unfeasible. The idea is currently being implemented on top of an already implemented query language for XML native databases based on the DOM model.

The structure of the paper is as follows. The next section formalizes XML data structures, introduces main concepts of the Stack-Based Approach and presents its formalized OQL-like query language – SBQL. Section 3 presents our approach to updatable views. Section 4 includes examples manifesting the power of our method and its possible applications. Section 5 concludes.

2 Stack-Based Approach (SBA) for XML

XML is one of the most commonly used standards to represent data in the Web. More and more data are stored in XML or mapped to XML by means of wrappers from semi-structured documents, relational databases, object-relational and object databases. Popularity of XML is evident in commercial relational databases like Oracle, which includes integrated XML repository. This increasing amount of distributed information stored/presented in XML indicates a need for a method of their integration, filtering and searching. This is the role for a query language for XML and XML view mechanism. Below we present an XML query language that is based on SBA.

SBA is based on the assumption that query languages are a kind of programming languages. The approach is abstract and universal, which makes it relevant to a general object model, in particular to XML structures. SBA makes it possible to precisely define the semantics of query languages, their relationships with object-oriented concepts, with imperative programming constructs and with programming abstractions including procedures, functional procedures, views, modules, classes, methods, etc.

2.1 XML Object Store Model

XML data form a tree structure with nested tags representing a hierarchy of nested objects. XML tags can also have attributes, which formally can be treated similarly to nested XML documents. We extend the classical XML store model by introducing relationships (links) between objects. We present a formal model of XML document store, adapting it to the SBA terms below.

In SBA each object has the following components:

- Internal identifier (OID); identifiers cannot be directly written in queries and are not printable. Let I be a set of such internal identifiers.
- External name (introduced by a designer, programmer or user) used to access the object from a program. Let N be a set of such external names.
- Content that can be a value, a link, or a set of objects.

Let V be a set of atomic values, e.g. numbers, strings, blobs. Atomic values also 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 object $as \langle i, n, v \rangle$.
- Link object $as \langle i_1, n, i_2 \rangle$.
- Compound object $as \langle i, n, S \rangle$, where S denotes a set of objects.

The definition is recursive and allows creating compound objects isomorphic to XML documents with an arbitrary number of hierarchy levels. Relationships (associations) are modeled through link objects. In order to model collections, SBA does not impose the uniqueness of external names at any level of data hierarchy, like in XML. Although XML objects have no explicit identifiers (references) on the level of XML text files, they must appear in any form of parsed XML, e.g. in the DOM model. Some form of XML object identification is also present in XPath and Xlink utilities.

In the above definition we do not determine (and are not interested in) a particular method of internal identification of XML documents.

In SBA, objects populate an *object store*, which is formed by:

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

Example data store. We present an example XML object store in Fig. 1. It contains a fragment of a database describing a company. The company has two employees; one of them being a chief of the IT department.

<pre> <Company> <Department id=„1”> <dNo>1</dNo> <dName>Computer Science</dName> <loc>Elms St. 21</loc> <loc>Wall St. 11</loc> <employs pointer=„true”>2</employs> <employs pointer=„true”>3</employs> <boss pointer=„true”>3</boss> </Department> <Employee id=„2” sex=„male”> <eNo>123</eNo> <name>John Smith</name> <job>designer</job> <sal>2345</sal> <rating>5.7</rating> <works_in pointer=„true”>1</works_in> </Employee> <Employee id=„3”> <eNo>456</eNo> <name>George Cooper</name> <job>consultant</job> <sal>7000</sal> <rating>3.1</rating> <works_in pointer=„true”>1</works_in> <manages>1</manages> </Employee> </Company> </pre>	<p>Objects:</p> <pre> <i₁, Company, { <i₂, Department, { <i₃, attr_id, 1>, <i₄, dNo, 1>, <i₅, dName, „IT”>, <i₆, loc, „Elms St. 21”>, <i₇, loc, „Wall St. 11”>, <i₈, employs, i₁₁>, <i₉, employs, i₂₀>, <i₁₀, boss, i₂₀> }>, <i₁₁, Employee, { <i₁₂, attr_id, 2>, <i₁₃, attr_sex, „male”> <i₁₄, eNo, 123>, <i₁₅, name, „JohnSmith”>, <i₁₆, job, „designer”>, <i₁₇, sal, 2345>, <i₁₈, rating, 5.7>, <i₁₉, works_in, i₂> }>, <i₂₀, Employee, { <i₂₁, attr_id, 3>, <i₂₂, eNo, 456>, <i₂₃, name, „George Cooper”>, <i₂₄, job, „consultant”>, <i₂₅, sal, 7000>, <i₂₆, rating, 3.1>, <i₂₇, works_in, i₂>, <i₂₈, manages, i₂> }> }> </pre> <p>Roots: { i₁, i₂, i₁₁, i₂₀ }</p>
---	--

Fig. 1. The object store

Note that there is no problem to map XML into the presented object store - it is isomorphic to XML. We treat attributes of an object as its internal objects and we distinguish them from regular internal objects by name prefix “attr_”. We identify link objects in XML by the attribute *pointer* set to true; for short, we do not present explicitly attr_pointer objects, but only parsed version in the SBA convention. In our example roots for the store refer to the *Company* object and to *Department* and *Employee* sub-objects. This is an arbitrary choice. Alternatively, we could assume that roots are

$\{i_2, i_{11}, i_{20}\}$ or there is a single root $\{i_1\}$. Our assumption means that queries can start from *Company*, *Department* and *Employee*.

2.2 Environment Stack (ES)

ES is one of the basic data structures in programming languages semantics and implementation. It supports 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 use. The stack makes it possible to associate parameters and local variables to a particular procedure (function, method, etc) invocation. Thus, safe nested calls of procedures from other procedures are possible, which includes recursive calls.

ES consists of *sections* that are sets of *binders*. A binder relates a name with a run time entity. Formally, it is a pair (n, x) , where n is an external name ($n \in N$) and x is a reference to an object ($x \in I$); such a pair is written as $n(x)$. We refer to n as the *binder name* and to x as the *binder value*. The concept of a binder can be generalized – x can be an atomic value, a compound structure, or a reference to a procedure/method.

The process that determines the meaning of a name is called *binding*. Binding follows the “search from the top” rule i.e. to bind a name n the mechanism is looking for the ES “visible” section that is the closest to the top of ES and contains a binder with the name n . If the binder is $n(x)$, then the result of the binding is x . To cover bulk data structures of the store model, SBA assumes that binding can be multi-valued, that is, if the relevant section contains several binders whose names are $n: n(x_1), n(x_2), n(x_3), \dots$, then all of them contribute to the result of the binding. In such a case the binding of n returns the collection $\{x_1, x_2, x_3, \dots\}$.

At the beginning of a session ES consists of a base section that contains binders to all database root objects. Other base sections can contain 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 (i.e., no calls of updating methods), the final ES state is exactly the same as the initial one. Fig.2 presents this idea for the evaluation of the query “*Employee* where <predicate containing name p >” for the object store from Fig.1. The presented intermediate state concerns the binding of name p occurring in the *where* clause of the query; the search order is presented by the arrows.

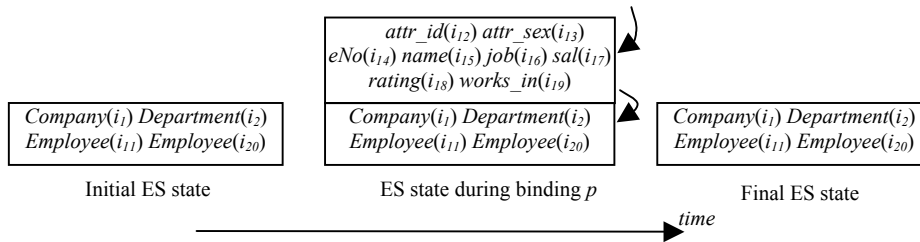


Fig. 2. States of ES during query evaluation

2.3 Stack-Based Query Language (SBQL)

In this section we present the general idea of SBQL, which is described in detail in [SBMS95, SKL95]. The language is implemented in the Loqis system [SMA90, Subi91], in the prototype of an XML query language for the DOM model and in the prototype for the ICONS project.

We argue that queries addressing XML need not be expressed through XML syntax, as suggested in [CFMR01]. There is no point in perceiving queries as XML files. In our opinion, XML syntax makes queries illegible. Legibility of queries is much more important for users than any other feature related to query syntax.

SBQL syntax. SBQL is based on an abstract syntax and the principle of *compositionality* - it syntactically separates query operators. The syntax of the language is as follows:

- A single name or a single literal is an (atomic) query e.g. *Employee*, *y*, “Smith”, 2.
- If q is a query and σ is a unary operator (e.g. sum, 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.

SBA is based on the assumption of operator orthogonality – we can freely combine operators unless it violates some type constraints. SBQL divides operators into two categories: algebraic and non-algebraic. Now, we shortly present these two kinds.

Algebraic operators. The operator is algebraic if it does not modify ES. Algebraic operators include string comparisons, Boolean and numerical operators, aggregate functions, set, bag and sequence operators and comparisons, the Cartesian product, etc. The evaluation of an algebraic operator is very simple: let $q_1 \Delta q_2$ be a query that consists of two subqueries connected by a symbol Δ denoting a binary algebraic operator Δ . First, q_1 and q_2 are evaluated independently; then Δ is performed on two partial results (taking them in the proper order), returning the final result.

Non-algebraic operators. Non-algebraic operators include *selection*, *navigation*, *dependent join*, *quantifiers*, etc. 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. This is the reason for which these operators are called “non-algebraic”: they do not follow the basic property of algebraic expressions evaluation, which requires independent evaluation of q_1 and q_2 .

The query $q_1 \theta q_2$ is evaluated as follows. For each element r of the query value (q_value) returned by q_1 , the subquery q_2 is evaluated. Before each such evaluation ES is augmented with a new section determined by r . After the evaluation, the stack 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 kind of combination depends on θ . Next, these partial results are merged into the final result.

New ES section(s) are constructed and returned by a special function *nested*. For an element r the function is defined as follows:

- If r is a single identifier of a complex object, e.g. i_{11} , then *nested* returns binders to attributes (sub-objects) of a given object.
- If r is an identifier of a link object, e.g., i_8 , then *nested* returns binders to the object the link points to (e.g. for i_8 it is $\{Employee(i_{11})\}$).
- If r is a binder, then *nested* returns $\{r\}$ (the set consisting of the binder).
- For r being a structure $struct\{r_1, r_2, r_3, \dots\}$ *nested* returns the sum of results returned by *nested* for r_1, r_2, r_3, \dots
- For other r *nested* returns the empty set.

Query Results. We call values returned by SBQL queries *q_values* and we define them in the following, recursive way:

- Each atomic value (e.g. 3, "Smith", TRUE, etc.) is a *q_value*.
- Each reference to an object (of any kind, e.g. $i_{11}, i_{12}, i_{16}, i_{18}$, etc.) is a *q_value*.
- If v is a *q_value* and n is any name, then a binder $n(v)$ is a *q_value*.
- If v_1, v_2, v_3, \dots are *q_values* and *struct* is a structure constructor, then $struct\{v_1, v_2, v_3, \dots\}$ is a *q_value*. In general, the order of elements in the structure is essential. This constraint can be relaxed if all v_i are binders. This construct generalizes a tuple known from relational systems.
- If v_1, v_2, v_3, \dots are *q_values* and *bag*, *sequence*, ... are collection constructors, then $bag\{v_1, v_2, v_3, \dots\}$, $sequence\{v_1, v_2, v_3, \dots\}$, ... are *q_values*.
- There is no other *q_values*.

Note that there is no problem to map final result of our query to XML, providing all its sub-results are binders. However, it makes no sense to map intermediate query results to the XML format, because our semantics requires returning references to objects that are lost after such a conversion.

ES is closely related to definition of semantics. SBA uses another, auxiliary stack QRES (Query REsult) for storing results of (sub)queries in the operational style of semantics. In other definitions (e.g. denotational) QRES is not necessary.

Example queries in SBQL³:

```
Company.Employee
Employee where job = "designer"
(Department where count(employs) > 20).boss.Employee.name
```

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. It also means that there

³ Note that our queries are more "diabetic" than SQL and query languages for XML (XQL, Xquery, etc.): there is no "select", "from" and other sugar. This essentially improves orthogonality, compositionality and readability of nested queries. The sugar can be of course freely added to SBQL, according to taste.

are no restrictions on calling functions within the body of (other) functions, what enables among others recursive calls. In current implementation procedures (and other features of SBQL) are untyped, but another group in our team intends to introduce static and dynamic type checking.

Below, we present an example function *underpaid* returning identifiers of employees whose salary is less than average and professions are listed in parameter *JobPar* (*JobPar* is a *call-by-value* parameter). Function has an auxiliary local variable *a*:

```
function underpaid ( in JobPar ) {  
  local a := avg ( Employee . sal );  
  return Employee where job = JobPar and sal < a; }
```

We can call the function in a query that returns names of departments where underpaid clerks work:

```
underpaid( "clerk " ) . works_in . Department . dName
```

Practically, in all the classical approaches (relational, object-relational and object-oriented) a view is essentially a functional procedure that is stored in a database. View updates are performed through side effects of view definitions, usually by various kinds of references to stored data (TID-s, OID-s, etc.) returned by view invocations. The art of view updating is focused on forbidding updates that may violate user intention, c.f. view updatability criteria, such as no over-updating of a view. For instance, to avoid over-updating in Oracle a user is allowed to update a virtual relation being a join of two stored relations, but updates can concern only attributes coming from the relation being on the foreign key side of the join. We definitely abandon this approach, disallowing any updates through side effects of view definitions. Instead, we introduce explicit information on intents of view updates in the view definition.

3 Updatable Views for XML

We formulated the following basic assumptions for our view mechanism:

- View definitions are complex entities in a spirit of abstract data types (but essentially different from ADTs), which define the information content of virtual objects together with all the required view updating operations. The operations overload generic updating operations on virtual objects and perform updates on stored data. The language for defining a content of virtual objects and view updating operations is based on SBQL and has full computational power of a programming language.
- Any view updating operation is fully in hands of a view definer. We assume no updating through side effects, e.g. by references returned by a view invocation.
- Preservation of the programming languages' principles, such as semantic relativism, orthogonality and no exceptional or special cases. These principles support universality, conceptual simplicity, simplicity of the use, easy implementation and optimisability of the view mechanism. They also much reduce the size of documentation and learning time. Semantic relativism is an essential novelty of our view definitions, not present in previous approaches to views. It means that view definitions can be nested with unlimited number of nesting levels, up to "atomic" view definitions which define atomic virtual data. If a virtual object has attributes, each

of them must be defined as a sub-view. Independently of the view hierarchy level, each view definition has the same syntax, semantics and pragmatics.

- Full transparency of virtual objects. When a view is defined, a user will be unable to recognize any difference in querying and manipulating virtual and stored objects.
- Universality. The approach that we propose will be applicable not only to XML-oriented databases, but also to object-oriented and relational databases. The approach can be easily extended to any kind complex object/data models.
- Queries involving views will be optimizable using query modification technique. Within the technique we can use all the optimization methods that have been developed for the given model (rewriting, indices, etc.).

In comparison to classical views (cf. SQL) we assume that the name of a view definition is different from the name of virtual objects determined by the view. Therefore, we explicitly introduce the *managerial name* of a view (used for operations on view definition) and a name of virtual objects (used for operations on virtual objects). We assume a simple naming convention where a managerial name has always the suffix “*Def*”, e.g. *RichEmpDef*.

Now, we present main concepts of our updatable views mechanism. More detailed description can be found in [KLPS02].

3.1 View definition

All current approaches to views take it for granted that the mapping between stored and virtual data/objects is determined by a single query. We consider such an approach inappropriate if one wants to introduce operations of view updates, because it causes a loss of semantic information. For instance, if a view returns (non-unique) boss names and average salaries in departments, then association of particular values returned by the view with particular departments can be lost; therefore, it would be impossible to write a view updating procedure that will require references to departments as parameters. Additionally, in this way we can take full control on any access to a view, including retrieval. A single query approach is a disadvantage of the (Oracle, SQL Server) “instead of” trigger views, making a lot of view updates impossible.

Therefore, we propose a two-query paradigm to view updates. The first query preserves all the necessary information about the stored source objects and the second query (wrapped in a procedure) takes the result of the first query as input and delivers the final mapping between stored and virtual objects. The first query returns a collection containing elements called seeds. A seed is used by the second query for making up a virtual object; the number of virtual objects is the same as the number of seeds. A seed is also used as a parameter of the updating procedures defined by the view definer for determining view updates. Passing this parameter is implicit (it is internal to the proposed mechanism). An entire virtual object growing up from a seed contains data determined by the second query, sub-views and the defined updating operations. There are no limitations concerning the complexity of a seed – it can be the result of queries involving joins or other query operators.

The result of the first query is the basis of the following operations:

- *Dereferencing* (the second query) that returns the value of a virtual object. This value can be complex: it can be composed of references, atomic values and names.

It must be (usually implicitly) applied in situations where an identifier must be changed to value, e.g. the context of such algebraic operators as +, <, sum, call-by-value parameters, etc.

- *Updating*. The operation performs assignment to a virtual object. It has an r-value as a parameter.
- *Deleting*. The operator performs deleting of a virtual object.
- *Inserting*. The operator performs inserting a new (virtual) object to the inside of a virtual object. It has a reference to a new object as a parameter.

The syntax of a view definition is illustrated by the following example:

```

create view RichEmpDef {
  virtual objects RichEmp { return (Employee where salary > 1000) as r }
                                     // generating the collection of seeds
  on_retrieve do { ..... };           // dereferencing
  on_update rvalue do { ..... };    // assignment
  on_delete do { ..... };             // deleting
  on_insert objectref do { ..... }; // inserting
  .... //further text of the definition
}

```

Keywords **on_retrieve**, **on_update**, **on_delete**, **on_insert** are identical for all definitions of views. Each of the clauses is treated as a procedure. Names of parameters such as *rvalue* and *objectref* can be chosen by the view definer.

When a view is defined it is considered as a regular store object. The database section of the ES contains both: the binder with the reference to view definition (with managerial view name) and the second binder with virtual objects name (to indicate the existence of virtual objects). Both these binders are necessary. The first one is required to use or change the view's definition and the second one allows querying and updating virtual objects.

3.2 Virtual identifiers

An essential issue concerns how to pass the information concerning an updating of a virtual object to the proper updating procedure defined within the view. Indeed, if a view invocation returns the result identical to the result of a query, then while performing the operation, the system is unable to recognize that the update concerns the view rather than regular stored objects. If the system cannot recognize it, it is unable to call the procedure that overloads the operation. The problem has been solved by the concept of *virtual identifier*, which is a triple:

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

Virtual identifiers are counterparts of object identifiers. They are constructed in such a way that they deliver all the necessary information on virtual objects to view updating procedures written by the view definer; the system knows that the update concerns a view (due to the flag "I am virtual"), which view (due to the *View definition identifier*) and which virtual object (due to the *Seed*).

Processing of virtual identifiers. In SBA each non-algebraic operator processing an identifier pushes on ES all the binders referring to the interior of the object having this identifier. If an identifier is a virtual one of the form $\langle \text{Flag } "I \text{ am virtual}", \text{view_def_id, seed} \rangle$, then ES is first augmented by the section containing $\text{nested}(\text{seed})$ and next, by another section containing binders to all subviews of the view_def_id view. In such a way we pass the seed parameter to all the (dereferencing, updating) procedures that are defined within the view and, at the same time, we make all sub-views (i.e. virtual attributes or sub-attributes) available for querying.

In Fig. 3 we present a general schema showing the control flow between different agents participating in serving view updates. Roughly, the scenario consists of:

1. Evaluation of a query invoking a view, which returns virtual identifiers.
2. Passing the virtual identifiers to the updating statement (containing the query).
3. Processing of the updating operation by the query/program interpreter. It recognizes that the operation concerns a virtual identifier. Thus, it makes no action on stored data, but passes the control to the proper procedure from the view definition. The interpreter prepares parameters for this procedure on the basis of seed stored within the virtual identifier.
4. Execution of the procedure, with an effect on stored data.
5. The control is passed back to the user program

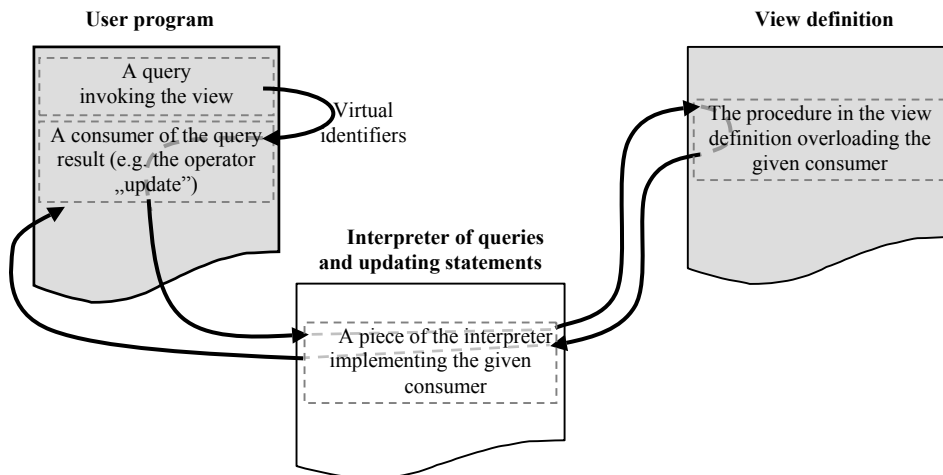


Fig. 3. Control flow during a view update

3.3 Nested views

In our approach we can nest views, what implies that we have to extend a notion of virtual identifiers in order to access all seeds of its parent virtual objects along the path of nesting. A possible extended form of a virtual identifier is the following:

$$\langle \text{Flag } "I \text{ am virtual}", (\text{View definition identifier}_1, \text{Seed}_1), \dots, (\text{View definition identifier}_n, \text{Seed}_n) \rangle$$

where (*View definition identifier*₁, *Seed*₁) refers to the most outer view, (*View definition identifier*₂, *Seed*₂) refers to its sub-view, etc., and (*View definition identifier*_{*n*}, *Seed*_{*n*}) refers to the currently processed view.

In a case 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 $nested(Seed_1) \cup nested(Seed_2) \cup \dots \cup nested(Seed_n)$, and then calls the proper procedure. This is the way of passing information on seeds to all the procedures. The call requires identification of the proper procedure within nested views, e.g. according to the expression: *View definition identifier*_{*n*}. *on_update*

3.4 View Parameters and Recursive Views

In our approach views, similarly to procedures or functions, can have parameters. Parameters concern only a procedure generating seeds. In this paper we do not assume that a parameter will be available for definitions of updating procedures (it requires some minor extensions to the mechanism). A parameter is a query, with no restriction. In particular, it may return $bag(r_1, r_2, \dots, r_n)$, where r_1, r_2, \dots, r_n are *q_values*. A query interpreter determines a method of parameters passing basing on view definition syntax. We are going to implement in our views the method of parameters passing known as *strict-call-by-value*, which combines *call-by-value* and *call-by-reference*. It means that the result of a query being a parameter is without any change passed to the function's body. Technically, it means that if for a formal parameter *par* the query returns the result *r* (however complex), then the corresponding activation record for function *f* is augmented by the single binder *par*(*r*). In this way the result of the query becomes available within the body of the function under name *par*.

Recursive views are side effects of SBA and its mechanisms. As shown, views are programming entities like functions or 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.

3.5 Optimization

Due to full orthogonality and consistent conceptual continuation (with no anomalies and irregularities), queries involving views are fully optimizable through the query modification technique [Ston75], as presented in [SuPl01]. The optimization concerns cases when the procedure defining a collection of seeds is reduced to single (however complex) queries, with no parameters and recursion. These conditions are satisfied for majority of views. In all such cases textual substitution of the views invocation by the corresponding query from the procedure defining seeds results in a semantically equivalent query that can be optimized by standard methods, e.g. by removing dead sub-queries, factoring out independent sub-queries and low level techniques (e.g. based on indices). An example view optimization process can be found in [KLPS02].

4 Examples

In this section we present examples illustrating power of our view approach. Examples use source of XML data presented in Fig. 1.

Example 1: Moving an employee to another department

For each employee the view *EmpBoss(Surname, BossSurname)* returns a virtual object containing a pair of strings: the surname of an employee and the surname of their boss. The view should facilitate the following operations: (1) updating a boss name to *NewBoss* that causes moving the employee to the department managed by the *NewBoss*; (2) deleting of a virtual object that causes deleting the corresponding employee. Dereferencing procedures cause returning proper strings rather than references. No other operation is supported.

Note that we consciously update a view in the way, which according to updatability criteria of Oracle is forbidden (updating on the primary key side of the join). Because in our case such updating is reasonable, it is allowed.

```
create view EmpBossDef {
  virtual objects EmpBoss { return Employee as e }
  on_delete do { delete e }
  create view SurnameDef {
    virtual objects Surname { return e.name as es }
    on_retrieve do { return es ^ "" } } //concatenation with the empty string
  create view BossSurnameDef { // enforces dereferencing
    virtual objects BossSurname {
      return e.worksIn.Department.boss.Employee.name as bs }
    on_retrieve do { return bs ^ "" }
    on_update ( NewBoss ) do {
      e.works_in :=& Department where (boss.Employee.name) = NewBoss } } }
```

We assume automatic updating of twin pointers by the system, c.f. the ODMG standard: e.g. updating of *works_in* causes automatic updating of the twin *employs*.

Examples of view calls:

1. Get surnames of all employees working for Smith:
(*EmpBoss where BossSurname = "Smith"*). *Surname*
2. Fire the employee named Cooper:
delete *EmpBoss where Surname = "Cooper"*;
3. Introduce a new boss Smith for employee Cooper:
for each *EmpBoss where Surname = "Cooper" do BossSurname := "Smith"*;
4. Change the surname of the employee named "Coper" to "Cooper":
for each *EmpBoss where Surname = "Coper" do Surname := "Cooper"*;
! Incorrect: *SurnameDef* does not support the updating operation.

Example 2: Updating average salaries in departments

A view *DeptAvgSal(DeptName, AvgSal)* returns a virtual object containing names and average salaries for all departments located in Warsaw. Updating the average sal-

ary causes distribution of the rise among all employees of the department proportionally to their previous salaries and to their assessment. Of course, there could be many other intentions of view update; each requires a specific *on_update* procedure.

```

create view DeptAvgSalDef {
  virtual objects DeptAvgSal{
    return (Department where loc="Warsaw") as d; }
  create view DeptNameDef {
    virtual objects DeptName { return d.dName as dn; }
    on_retrieve do { return dn ^ "" ; } }
  create view AvgSalDef {
    virtual objects AvgSal{ return avg( d.employs.Employee.sal ) as a; }
    on_retrieve do { return a; }
    on_update ( newAvgSal ) do {
      create local weightedSum := sum( d.employs.Employee.(sal*rating));
      create local ratio := (newAvgSal - a) * count( d.employs ) / weightedSum;
      for each d.employs.Employee do sal := sal + ratio*sal*rating; } } }

```

Using this view we can change average earnings in departments in Warsaw e.g. rise the average earnings in the Toys department by 100:

```

for each DeptAvgSal where DeptName = "Toys" do AvgSal:= AvgSal+ 100;

```

Note that the distribution of individual updates is proportional to earnings and ratings, but the final effect is that the average salary rise in the Toys department is 100. The example illustrates the power of our method: we are able to perform view updating operations considered earlier by many professionals as absolutely unfeasible. We show here that such updates are not only feasible, but could be reasonable and necessary for many applications, which require complex mappings of business ontologies.

5 Conclusion

We have presented a new approach to updatable views, which is based on the Stack-Based Approach. We introduced the view mechanism for XML, but the approach can be easily extended to more complex object oriented databases containing notion of classes, inheritance and dynamic roles. Currently, we are implementing the ideas on top of an already implemented query language SBQL for an XML native database. Afterwards, we are going to extend the implementation to cover more complex object store models, in particular some superset of the RDF data model. We have shown that our approach enables the users to define very powerful views, including views with parameters, with a local environment, recursive, with side effects, etc. The concept is consistent, relatively easy to implement, very simple to use and enabling optimization by powerful query modification methods.

XML views may have many practical applications in the Web context. Views with full computational power, as described in this paper, cover also various kinds of mediators, wrappers and adaptors. The idea of such powerful views is at least worth discussion in the database and Web communities.

References

- [AAC+98] S. Abiteboul, B. Amann, S. Cluet, T. Milo, and V. Vianu. Active views for electronic commerce. Conf. sur les Bases Données, 1998.
- [Abit00] S. Abiteboul. On Views and XML. Proc. of PODS Conf., 1999, 1-9
- [AGM+97] S. Abiteboul, R. Goldman, J. McHugh, V. Vassalos, and Y. Zhuge. Views for Semistructured Data. Proceedings of the Workshop on Management of Semistructured Data, Tucson, Arizona, May 1997
- [CFMR01] D. Chamberlin, P. Fankhauser, M. Marchiori, J. Robie. XML Query Requirements. W3C Working Draft 15, February 2001
- [KL02] H. Kang, J. Lim: Deferred Incremental Refresh of XML Materialized Views. CAiSE 2002: 742-746
- [KLPS02] H. Kozankiewicz, J. Leszczyłowski, J. Płodzień, and K. Subieta. Updateable Object Views. Institute of Computer Science of PAS, Report 950, 2002
- [Lac01] Z. Lacroix. "Retrieving and Extracting Web data with Search Views and an XML Engine". Workshop on Data Integration over the Web, in conjunction with the 13th CAiSE Conf., Switzerland, 2001.
- [LAW99] T. Lahiri, S. Abiteboul, and J. Widom. Ozone: Integrating Structured and Semistructured Data. Proceedings of the 7th Intl. Conf. on Database Programming Languages, Kinloch Rannoch, Scotland, 1999
- [SBMS95] K. Subieta, C. Beeri, F. Matthes, and J. W. Schmidt. A Stack Based Approach to Query Languages. Proc. of 2nd Intl. East-West Database Workshop, Klagenfurt, Austria, 1994, Springer Workshops in Computing, 1995.
- [SKL95] K. Subieta, Y. Kambayashi, and J. Leszczyłowski. Procedures in Object-Oriented Query Languages. VLDB Conf., 182-193, 1995
- [SMA90] K. Subieta, M. Missala, and K. Anacki "The LOQIS System. Description and Programmer Manual", Institute of Computer Science of PAS, Report 695, 1990
- [Ston75] M. Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. Proc. of SIGMOD Conf., 1975
- [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
- [SuPI01] K. Subieta, J. Płodzień. Object Views and Query Modification. In Databases and Information Systems (eds. J. Barzdins, A. Caplinskas), Kluwer Academic Publishers, 2001, 3-14
- [Wied92] G. Wiederhold. Mediators in the Architecture of Future Information Systems, IEEE Computer Magazine, March 1992