SBQL Object Views - Unlimited Mapping and Updatability*

Radosław Adamus[1,2], Krzysztof Kaczmarski[1,3], Krzysztof Stencel[1,4], Kazimierz Subieta[1,2]

[1] Polish-Japanese Institute of Information Technology, Warsaw, Poland
[2] Computer Engineering Department, Technical University, Łódź, Poland
[3] Faculty of Mathematics and Information Science, Warsaw University of Technology, Warsaw, Poland
[4] Institute of Informatics, Warsaw University, Warsaw, Poland
r.adamus@kis.p.lodz.pl, k.kaczmarski@mini.pw.edu.pl, {stencel, subieta}@pjwstk.edu.pl

Abstract. The paper presents updatable object views defined in SBQL. They are unlimited in two important dimensions. First, because they are defined in a query/programming language with the full algorithmic power, they allow for any mapping of stored data onto virtual one. Second, the mechanism does not restrict updates of virtual data. 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. Again, he/she has the full algorithmic power at hand. We follow the Stack-Based Architecture, a theory of object-oriented query languages based on the classical concepts of programming languages, such as environment stack and naming-scoping-binding ideas. The described mechanism has been successfully implemented an tested in ODRA, a prototype object-oriented DBMS.

1. Introduction

Database views[1] are an important feature of relational databases and SQL. They are also the subject of various research, both theoretical and practical [1, 2, 3, 4, 5][2], in particular, in the context object-oriented databases, object-relational databases [6] and XML technologies [7]. For object-oriented databases there are implemented prototypes such as views in O_2 [8] and ActiveViews [9]. Views occur also in the ODMG standard [10] in the form of the "define" statements of OQL. The SQL:99 standard [11] devotes to views a lot of attention: views are specified and discussed on more than 100 pages of the document.

Views present no significant conceptual problem for retrieval. In this role a view can be considered a programming function returning a bulk output (e.g. a relation). Because such functions (in SQL) have no local variables (a view is determined by a single query), they are semantically equivalent to macro-definitions stored at the server side. This implies a simple and efficient technique of optimization of queries invoking views, known as *query modification* [12]. The technique is based on macro-substitution: each view invocation occurred in a query is macro-substituted by the view body; then such an expanded query is optimized. In this way a view is not materialized during invocation which results in substantial performance gain. In SQL (due to its non-orthogonality and re-naming of columns) the macro-substitution requires non-trivial algorithms. In SBQL [13] the macro-substitution is just a simple operation on abstract syntax trees. Optimization of a resulting expanded query presents some issue, because usually a view delivers much more data than it is required in a particular query invoking the view. Hence, after macro-substitution, the resulting query contains unnecessary parts (so-called *dead subqueries*) that should be removed to improve the performance. In a general case, removing dead subqueries requires a sophisticated algorithm [14].

* This work is supported by European Commission under the 6th FP project e-Gov Bus, IST-4-026727-ST
[1] In his paper we do not deal with materialized views that are actually database snapshots rather than views.
[2] There are more than 500 papers devoted to views, hence we limit our citations to just a few.

In contrast to retrieval, updating of virtual data delivered by a view leads to serious conceptual and implementation challenges. In our opinion (before SBQL) no proposal concerning view updating presents a satisfactory idea how to cope with the problem (known since 1974 [15]). It attracted many researchers and is quite well documented by many papers and examples. The problem is that updates of virtual data must be mapped into updates of stored data in a consistent way, without warping the intention of the user. However, such a mapping of updates can be done in many ways (sometimes, in infinitely many ways) and the system is unable to choose automatically which of them would satisfy the user intention. There are many theoretical proposals how to cope with the problem, but in our opinion they present too little progress (and are not implemented for the wide practical use). As an opposite trend, instead of solving the view updating problem, some authors propose to limit the cases when view updating is allowed by *view updatability rules*; all "dangerous" cases are to be rejected by the system [16]. Unfortunately, the danger is so common that for object-oriented models and their query languages only quite straightforward views are enough safe. This severely reduces expectations and promises behind the idea of views.

The first reasonable (and practically proven) idea to solve the view updating problem is based on so-called *instead of* triggers implemented in Oracle, SQL Server and DB2. If the system recognizes that an update concerns a virtual table, then instead of a regular updating action the system executes a special trigger (*instead of* trigger) containing a code written by the view definer. The code precisely determines the intention of the view update and it can do any action on a database and its environment to satisfy the intention. In this paper we are not able to present details of the method, but only summarise that it opens a right door to solve the view updating problem. Note that the problem is solved on the ground of programming languages (triggers are merely programs) rather than on the ground of classical theoretical concepts such as the relational algebra, calculus, mathematical logic, etc.

Although *instead of* triggers open a new dimension for view updates, they are still limited. The limitations concerned four aspects: (1) the datamodel is relational rather than object-oriented; (2) the power of the view definition mechanism is limited by SQL, which is much below the full algorithmic power; (3) strong static type checking is absent in SQL thus in any feature based on SQL; (4) possible performance problems: *instead of* triggers require materialization of views after invocations.

In our approach to object-oriented views we did not follow *instead of* trigger views. Our idea is based on our knowledge on programming languages' semantics and compilers and it 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 original view updates addressing virtual objects. The procedures are defined on top of the query language SBQL. The view definer should write an appropriate procedure for each of necessary view updating operations. Queries involving such views are then optimisable through the query modification technique, and then, by other optimization techniques based on rewriting rules, indices, caching, etc.

The approach has obvious consequences. A 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 Architecture (SBA), a new theory of

query languages [14, 17, 18] addressing, in particular, object-oriented models with any level of sophistication. The approach has roots in the semantics of programming languages. It integrates query languages and programming languages into a unified, consistent and non-redundant system of notions.

The most important property of views is transparency, which means that the user formulating a query needs not to distinguish between stored and virtual data. The SBQL updatable views present the first in the IT history universal, consistent and implemented solution of the view updating problem for object databases (actually, for any kind of databases). The method is much more general than *instead_of* triggers. Our solution for updatable views is supported by sophisticated query optimisation methods, which are not applicable to other proposals concerning updatable views. The approach has already been implemented in a number of research prototypes. The last and the most complete one is ODRA (Object Database for Rapid Application development) [19].

SBQL views are thoroughly described in the PhD thesis [20]. From that time, they have undergone a number of modifications and improvements due to the experiences gained in the implementation of the very mechanism and the conclusions drawn from elaboration of complex view examples both in centralized and distributed settings.

The rest of the paper is organized as follows. Section 2 presents the general idea of SBQL views. Section 3 introduces the key notion of seeds of virtual objects. Section 4 describes the way to define overridden meanings of operations on virtual objects. Section 5 discusses sub-views (corresponding to sub-objects). Section 6 explains the idea of virtual pointers. Section 7 presents stateful views. Section 8 is a short note on the quite new idea of overloading views. Section 9 concludes.

2. General Idea of SBQL Updatable Views

The idea of SBQL updatable object views relies in augmenting the definition of a view with the information on users' intents with respect to updating operations. An SBQL updatable view definition is subdivided into two parts. The first part is the functional procedure, which maps stored objects into virtual objects (similarly to SQL, but with full algorithmic power). The second part contains redefinitions of generic operations on virtual objects. These procedures express the users' intents with respect to update, delete, insert and retrieve operations performed on virtual objects. A view definition usually contains definitions of sub-views, which are defined on the same rule, according to the relativity principle. Because a view definition is a regular complex object, it may also contain other elements, such as procedures, functions, persistent objects, etc.

A view definition deals with two names. The first one is a managerial name that can be used to perform administration operations on the view definition, for instance, delete it, insert an object into it, etc. The second name is the name of virtual objects that are delivered by the view. The managerial name is optional. If it is not specified it is assumed by default that the managerial name will be the name of virtual objects suffixed with the string "*Def*". However, such a simple rule does not work in all cases (e.g. for overloading views [21]).

3 Seeds of Virtual Objects

In contrast to all existing approaches to views, an SBQL view does not return complete virtual objects as the result of a view invocation. This decision is motivated both by the new concept of views and by performance. Invocation of an SBQL view returns only *seeds* of its

virtual objects. A seed is a small piece of a virtual object that uniquely identifies it. The nature of seeds is not constrained, it can be simply a reference to an object, a value, a structure, etc. The view definer introduces seeds according to own will and requirements. The rest of a virtual object is delivered according to the need of an application that uses it. For instance, if a virtual object has a virtual attribute *address*, but an application does not use it, then *address* is not delivered. Seeds are also the conceptual basis for updating virtual objects: they parameterize updating operations that are specified by the view designer.

The first part of a view definition is a declaration of a virtual object. The declaration is similar to a variable declaration. It states the name, type and cardinality of virtual objects defined by the view. The second part of a view definition body has the form of a functional procedure named *seed*. The name of the virtual objects procedure is the name of virtual objects that the view returns. The *seed* procedure returns a bag of seeds. Seeds are then (implicitly) passed as parameters of procedures that overload operations on virtual objects (see: operators on virtual objects). Usually, seeds have to be named (i.e. they are binders), to identify them in the body of procedures. This is not obligatory if another identification method is possible. This name is then used in procedures that overload operators on virtual objects and within sub-views definitions.

Let us assume the following declaration of *EmpType* and *DeptType* types and *Emp* and *Dept* collections of objects:

```
type EmpType is record {
                    name: string;
                    deptName: string;
                    salary: integer;
                    opinion: string [0..1]; }
type DeptType is record {
                    dName: string;
                    location: string; }
Emp: EmpType [0..*];
Dept: DeptType [0..*];
```

The example below defines the view returning only those employees that earn more than 2000. The name of virtual objects is *RichEmp* and the managerial view name is *RichEmpDef*.

```
view RichEmpDef {
     virtual RichEmp : record {
                    name:string;
                    salary:integer;
                    worksIn: ref Dept;
               }[0..*];
     seed: record {e: ref Emp;}[0..*] {
               return (Emp where salary > 2000) as e;
     }
     // the rest of the view definition
}
```

First *RichEmpDef* view declares the virtual variable named *RichEmp* (our virtual objects). As we can see, virtual objects are structurally different from the *Emp* objects. *RichEmp* contains a name and a salary amount that are similar to those in *Emp* object. Instead of department name (*deptName* attribute) virtual object defines a (virtual) pointer to a *Dept* object. The *opinion* attribute is not visible through virtual object.

The second part is a definition of virtual objects' seeds. The *seed* procedure returns seeds of the declared type. In this case it returns named values (binders) that are represented as

structures with one element named e. A binder value is an id of an *Emp* object. The cardinality is same as the cardinality of the virtual variable.

From the programmer point of view (in his/her imagination) the presence of this view definition can be perceived as the database contains objects named *RichEmp*. If the view is properly and completely defined the application programmer has no programming option to distinguish virtual and stored objects (what is just the essence of the full transparency of views). A simple query `RichEmp` returns identifiers of virtual objects (so-called *virtual identifiers*), having seeds as main components. Currently, however, no operation on them is possible, because they have to be explicitly defined in the further part of the definition.

4. Operators on Virtual Objects

The operations that can be performed on virtual objects are defined in the second part of a view definition. They allow the programmer to create the behaviour of virtual objects in the context of the following generic operations:

- *retrieve:* (dereference) returns the value of the given virtual object;
- *update:* modifies the value of the given virtual object;
- *create/insert* : create a new virtual object, insert an object into a virtual object;
- *delete*: removes the given virtual object;
- *navigate*: navigates according to a virtual pointer.

If there is no definition of a particular operator inside a view definition it is assumed that the operation is forbidden for the virtual objects generated by the view. The definitions of the operators have procedural semantics. Each operator has a predefined name: *on_retrieve*, *on_update*, *on_new*, *on_delete* and *on_navigate* respectively.

The execution of given operator is implicit. If the system detects that the parameter of the operation is a virtual object, instead of taking system default action the appropriate view operator procedure is invoked (just like in *instead of* trigger views). A seed describing a virtual object is implicitly passed as a default parameter to the procedure through the environment stack. After the execution the control is passed back to the user program.

The above description is similar for all operators except the operator for creating virtual objects (*on_new*). By its nature, this operator cannot be executed in the context of a virtual object. The system passes the control to the *on_new* procedure if in the environment where the virtual objects with the given name are defined a new object with the same name appears (e.g. it was created by the *create* operator or inserted by the *insert* operator). The value of the object is passed as an argument to the *on_new* procedure. After the *on_new* procedure ends, the object is automatically deleted (i.e. a material object is substituted by a virtual object). The procedure *on_new* performs (determined by the view definer) actions on stored objects that result in the effect that the new virtual object appears in the database environment. To this end, new stored objects can be created in the database, but the questions which objects and how they are created depend on the current need that is to be recognized by the view definer.

Now we can extend the *RichEmpDef* view with the operators. Assume that we want to allow to perform all the operation on the virtual object except the deletion that will be forbidden. First we define the dereference operator (*on_retrieve*).

```
view RichEmpDef {
     virtual RichEmp : record {
                    name:string;
                    salary:integer;
                    worksIn: ref Dept; }[0..*];
     seed: record {e: ref Emp;}[0..*] {
               return (Emp where salary > 2000) as e;
     }
     on_retrieve {
       return e.name as name,
              e.salary as salary,
              ref (Dept where dName = e.deptName) as worksIn;
     }
     // the rest of the view definition
}
```

The *on_retrieve* procedure uses an implicit parameter *e* (the seed of a virtual object) to construct the required structure of a dereferenced virtual object (according to its type defined inside the view). Because binding to name of the seed returns a reference to an *Emp* object (see the seed procedure type) the programmer has full control on the transformation of regular object (*Emp*) to virtual object (*RichEmp*) (note that explicit *deref* operator can be omitted here because the system can add it implicitly basing on the return type). In more complex cases the operator can perform additional tasks (as it is a regular procedure).

Next we extend the virtual objects with update operator:

```
view RichEmpDef {
     virtual RichEmp : record {
                    name:string;
                    salary:integer;
                    worksIn: ref Dept; }[0..*];
     seed: record {e: ref Emp;}[0..*] {
               return (Emp where salary > 2000) as e;
     }
     on_retrieve {
       return e.name as name,
              e.salary as salary,
              ref (Dept where dName = e.deptName) as worksIn;
     }
     on_update {
       e.name := value.name;
       e.deptName := value.worksIn.dName;
       if(e.salary < value.salary) {
          e.salary := value.salary;
       }
     }
     // the rest of the view definition
}
```

As for *on_retrieve*, the *on_update* operator has an implicit parameter *e* that is the seed of a virtual object. The update operator has another parameter – the value that is to be assigned to a virtual object. The programmer needs not to define the name of this parameter assuming the

default name **'value'.** This is not a rule, this parameter can also be determined explicitly. The default **'value'** name can be used inside the operator procedure body. Note that the type of a parameter is (implicilty) the type of the virtual object (*RichEmp*) but we update the regular object (*Emp*). In other words, *on_update* operator maps virtual data onto regular (in contrary *to on_retrieve* operator that maps regular data onto a virtual one).

The semantics of sample update operation assumes that the name of a virtual *RichEmp* object is directly mapped to the name of a regular *Emp* object. Because the virtual object defines a *worksIn* pointer object, the value of name subobject of the target *Dept* object (the pointer points to) is mapped onto the *deptName* value. The definition assumes that the *salary* field value in *Emp* is updated only if is less that the salary value inside virtual object.

Finally we define *on_new* operator to allow creating new *RichEmp* virtual objects. Let assume the semantic that the object will be created only when the attribute salary will be greater than 2000.

```
view RichEmpDef {
     virtual RichEmp : record {
                    name:string;
                    salary:integer;
                    worksIn: ref Dept; }[0..*];
     seed: record {e: ref Emp;}[0..*] {
               return (Emp where salary > 2000) as e;
     }
     on_retrieve {
       return e.name as name,
              e.salary as salary,
              ref (Dept where dName = e.deptName) as worksIn;
     }
     on_update {
       e.name := value.name;
       e.deptName := value.worksIn.dName;
       if(e.salary < value.salary) {
          e.salary := value.salary;
       }
     }
     on_new newEmp {
       if(newEmp.salary > 2000)
          create permanent Emp(
               newEmp.name as name,
               newEmp.salary as salary,
               newEmp.worksIn.dName as deptName
          );
     }
     // the rest of the view definition
}
```

In this *on_new* definition the name of the parameter is given explicitly and can be used inside of the operator procedure body. As in case of *on_update* operator the *on_new* operator maps virtual data onto a corresponding regular one (note that the attribute *opinion* is optional, hence is not filled in).

5 Nested Views (sub-views)

According to the object relativity principle, each view can contains sub-views; hence each virtual object can contain virtual subobjects. The number of view nesting levels is unlimited. Syntax and semantics of nested sub-view definitions is the same as for enclosing views. If a virtual object has attributes they can be defined by sub-views. In principle, there is no way to declare attributes and sub-attributes of virtual objects in another way, but this is only temporary decision that can be changed. The type, name and cardinality of a nested virtual object have to conform to one of the field defined for the virtual object in enclosing view. Sub-views define their own seeds and *on_xxx* operators procedures. Additionally, the procedures have access to seeds generated by enclosing views.

The *RichEmp* virtual objects defined in the previous sections currently does not have any attribute (sub-object), even though it seed is based ob the complex *Emp* object. To define virtual attributes we need to define sub-views. In our example we can decide which *Emp* attributes become virtual attributes of *RichEmp*. For each attribute we can separately decide which operators will be available for it. Below a definition of an attribute *name* is shown:

```
view RichEmpDef {
     // declaration of objects, seeds and operators for RichEmp

     // subview
     view nameDef {
          virtual name:string;
          seed : record { n:string; }
          {
           return e.name as n;
          }
          on_retrieve { return n; }
     }
     // the rest of the view definition
}
```

The view *nameDef* introduces a virtual attribute *name* of *RichEmp* virtual objects. The RichEmp virtual object in the enclosing view declares name of type *string* and the default cardinality [1..1] as one of its fields. The declaration of nested virtual object conforms to this specification. The seed procedure for the *name* virtual object accesses the enclosing virtual object seed and returns a binder *n* with string representing the *RichEmp* name.

The view definition contains only one operator – *on_retrieve*. Therefore, operations other than dereference are forbidden for the *name* virtual attribute.

Similar approach has to be taken for the *salary* attribute. Of course the programmer can decide which operators are to be available for a given virtual attribute.

6 Virtual Pointers

Up to now virtual objects can be perceived by the user as simple objects (defined by the view without nested sub-views) or complex objects (defined by the view with nested sub-views). For completeness of the transparency we need also to define virtual entities that can be perceived as pointer objects.

A pointer object allows the programmer to navigate through an object graph. The unique property of SBQL is that the environment of a pointer object is represented by the binder

named with the name of pointed object; thus, navigation through the pointer object requires typing the name of the target object. This property allows us to separate the reference to pointer itself and the reference to pointed object. For example if we assume that *friend* is a pointer sub-object of *Person* object, then the following query will return the reference (or a bag of references) of pointer object named *friend*.

```
(Person where name = "Kim").friend
```

Such references can be the subject of imperative operations (e.g. updated, deleted). To return the references to objects pointed by the *friend* objects one must write:

```
(Person where name = "Kim").friend.Person
```

To define a virtual pointer with analogous semantics we need to introduce into the view definition a new operator. A virtual object acts as a virtual pointer if its definition is augmented by the operator *on_navigate*.

As usual, the operator is defined as a functional procedure. It must return a reference (or a virtual reference) of a "virtually" pointed object. As for the other operators its return type implicitly corresponds to the declared type of a virtual object. These two assumptions enforce that only those virtual objects which return reference to the other objects can possess *on_navigate* operator and be perceived as (virtual) pointers.

The operator procedure is implicitly executed during the process of calculating a nested environment in the context of a non-algebraic operator (see: non-algebraic operators). The result reference of a *on_navigate* call is then available within the virtual object environment (the semantics is the same as for regular pointer objects).

At this stage we'll introduce to the *RichEmp* virtual object the virtual pointer *worksIn* that will point at the department the given rich employee works in. The definition of the virtual pointer attribute requires addition of a suitable sub-view. To transform virtual object into virtual pointer we'll define the *on_navigate* operator. We also assume that the virtual pointer is a subject of dereference and update operation and define *on_retrieve* and *on_update* operators.

```
view RichEmpDef {
     // declaration of objects, seeds and operators for RichEmp

     // declarations of other subviews of RichEmp

     view worksInDef {
          virtual worksIn:ref Dept;

          seed :record{ dn:Emp.deptName; } {
               return e.deptName as dn; }

          on_navigate { return Dept where dName = dn; }

          on_retrieve { return Dept where dName = dn; }

          on_update { dn := value.dName; }
     }
     // the rest of the view definition
}
```

The *seed* procedure returns the reference to a *deptName* attribute inside an *Emp* object. The *on_navigate* operator returns a reference to a *Dept* object having *dName* equal to the *deptName* value. The result of *on_navigate* is a reference of a virtual pointer target object.

The sample code inside the *on_retrieve* operator procedure is the same as for the *on_navigate*. Both operators have to return the value of the type declared for the virtual object. But the code can perform some additional tasks different for navigating and updating.

The update semantics is straightforward. An argument is a reference of a *Dept* object. To change an employee's department we simply update the *deptName* attribute value with the value of the argument department name object.

7 Stateful Views

A view definition can include local objects. This is necessary for stateful views, which have a lot of applications, for example, to store security data or the state of network connections. The situation can be compared to instance and class invariants known from popular object-oriented programming languages. The sub-views define non-static (virtual objects) attributes and the local view objects are like static (class) attributes.

The definition of a view local objects is similar to declaring global or local variables but the declaration is placed inside the view definition.

We extend the *RichEmpDef* view with a state. Assume that the employee's richness level is parameterised by the database administrator. *RichEmp* virtual objects procedure will use this parameter to select those employees whose salary is greater than a parameter value. To do this we must introduce the state to the view and modify *RichEmp* virtual objects procedure (and all operators that depend on the earning threshold).

```
view RichEmpDef {
     virtual RichEmp : record {
                    name:string;
                    salary:integer;
                    worksIn: ref Dept;
               }[0..*];
     seed: record {e: ref Emp;}[0..*] {
               return (Emp where salary > threshold) as e;
     }
     on_retrieve {
       return e.name as name,
              e.salary as salary,
              ref (Dept where dName = e.deptName) as worksIn;
     }
     on_update {
       e.name := value.name;
       e.deptName := value.worksIn.name;
       if(e.salary < value.salary) {
          e.salary := value.salary;
       }
     }
     on_new newEmp {
       if(newEmp.salary > 2000)
          create permanent Emp(
               newEmp.name as name,
               newEmp.salary as salary,
```

```
                newEmp.worksIn.name as deptName
            );
        }
        // the rest of the view definition
        //declaration of the view local object
        threshold: integer;
}
```

The threshold object is local to the view definition and accessible through the view managerial name. Now the entitled user can change the threshold level with use of the following statement:

```
RichEmpDef.threshold := 2500;
```

Subsequent calls to this views will select rich employees according to this new threshold value and not any other value.

8 Overloading Views

In the PhD thesis [21] a new application of views to aspect-oriented databases is developed. The idea is that virtual objects named X transparently overload stored objects named X in such a way that any access or update of stored objects X must be done via virtual objects X. Within virtual objects X the view definer can program any additional action. This additional action is focused within one view definition instead of being dispersed among many places of application programs where X objects are accessed or updated. Thus, overloading views follow the motivation of aspect-oriented paradigms [22].

The idea of overloading views is an alternative to the database feature known as *triggers*. Overloading views, however, present a bit different quality than triggers and these two ideas can coexist in one system as options for programmers. Overloading views could be especially important for the application maintenance phase, when new features must be added to already working applications or when a database schema needs to be essentially changed.

9 Conclusion

In this paper we have shown the mechanism of updatable object views defined within the Stack-Based Architecture (SBA) and query/programming language SBQL. SBQL views have been successfully implemented in ODRA, an object-oriented DBMS. We have explained and exemplified a number of facilities which together constitute a very powerful view tool. It can be used to virtualize any content of a database. Indeed, virtual objects generated by SBQL views have all the features of ordinary stored objects. They can have attributes (i.e. subviews) and can be virtual pointers. They can also provide all updating operations with virtually any meaning defined by the view author by means of a language with full algorithmic power. SBQL can also be stateful. This opens a wide spectrum of applications: from atypical (one example shown in this paper) to more typical connected with distributed databases and peer-to-peer networks (the state of a views in such a setting could be e.g. the list of active neighbours and super-peers). Virtual objects can also be connected to classes and according to SBA this feature does not lead to conceptual or implementation problems.

References

[1] G.Guerrini, E. Bertino, B.Catania, J.Garcia-Molina. A Formal Model of Views for Object-Oriented Database Systems. Theory and Practice of Object Systems, 3(3), 1997, 157-183

[2] M.Gentile, R.Zicari. Updating Views in Object-Oriented Database Systems. Proc. of Intl. Symposium on Advanced Database Technologies and their Integration, Nara, Japan, 1994

[3] S.Heiler, S.Zdonik. Object Views: Extending the Vision. Proc. of 6-th Intl. Conf. on Data Engineering, 1990

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

[5] Subieta,K., M.Missala. View Updating Through Predefined Procedures. Information Systems 14(4), 1989, 291-305

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

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

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

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

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

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

[12] M.Stonebraker. Implementation of Integrity Constraints and Views by Query Modification. Proc. of SIGMOD Conf., 1975

[13] K.Subieta, J.Płodzień. Object Views and Query Modification, in: *Databases and Information Systems*, Kluwer Academic Publishers, pp. 3-14, 2001

[14] K.Subieta. *Theory and Construction of Object-Oriented Query Languages* (in Polish), PJIIT - Publishing House, 2004, 522 pages

[15] E.F.Codd. Recent investigations in a relational database systems. Information Processing, Vol. 74 (Proc. IFIP Congr. Stockholm), North Holland, Amsterdam, 1974, 1017-1021

[16] U.Dayal, P.A.Bernstein. On the Correct Translation of Update Operations on Relational Views. ACM Transactions On Database Systems Vol. 7, No 3, 1982

[17] K.Subieta, Y.Kambayashi, and J.Leszczyłowski. *Procedures in Object-Oriented Query Languages*. Proc. VLDB Conf., 182-193, 1995

[18] SBA and SBQL Web pages: http://www.sbql.pl/

[19] M.Lentner, K.Subieta: ODRA: A Next Generation Object-Oriented Environment for Rapid Database Application Development. Springer LNCS 4690, 130-140, 2007

[20] H.Kozankiewicz. *Updateable Object Views*, PhD Thesis, Institute of Computer Science, Polish Academy of Sciences, 2005, http://www.sbql.pl/phds

[21] R.Adamus. *Programming in Aspect-Oriented Databases*, PhD Thesis, Institute of Computer Science, Polish Academy of Sciences, 2005, http://www.sbql.pl/phds

[22] G.Kiczales et al.: Aspect-Oriented Programming. Proc. ECOOP Conf., Springer LNCS 1241, 220-242, 1997