*Journal of Universal Computer Science, vol. 12, no. 7 (2006), 938-957*
*submitted: 28/1/06, accepted: 6/4/06, appeared: 28/7/06 © J.UCS*

# Type-safe Versioned Object Query Language

**Rodrigo Machado, Álvaro Freitas Moreira, Renata de Matos Galante**
Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Cx. P. 15.064 – CEP 91.501-970 – Porto Alegre – RS – Brasil
{rma, afmoreira, galante}@inf.ufrgs.br

**Mirella Moura Moro**
Department of Computer Science & Engineering
University of California Riverside (UCR)
900 University Ave, Riverside, CA 92521 – USA
mirella@cs.ucr.edu

**Abstract:** The concept of versioning was initially proposed for controlling design evolution on computer aided design and software engineering. On the context of database systems, versioning is applied for managing the evolution of different elements of the data. Modern database systems provide not only powerful data models but also complex query languages that have evolved to include several features from complex programming languages. While most related work focuses on different aspects of the concepts, designing models, and processing of versions efficiently, there is yet to be a formal definition of a query language for database systems with versions control. In this work we propose a query language, named Versioned Object Query Language (VOQL), that extends ODMG Object Query Language (OQL) with new features to recover object versions. We provide a precise definition of VOQL through a type system and we prove it safe in relation to a small-step operational semantics. Finally, we validate the proposed definition by implementing an interpreter for VOQL.
**Key Words:** Operational Semantics, Object-oriented Database Management Systems, Type Systems, Query Languages.
**Category:** H.2.3, F.3.2

## 1 Introduction

The concept of versions was initially proposed for controlling design evolution and co-authoring on computer aided design [Katz 1990] and software engineering [Conradi and Westfechtel 1998, Westfechtel et al. 2001]. In those environments, versioning is applied to files, such that different alternatives or revisions of a document (e.g. source code, electronic models, product descriptions) are stored on different files. Hence, the versions are mostly handled by the operating system (we refer to [Robbes and Lanza 2005] for a more recent survey). Probably, the most common tools for handling this type of versions are CVS (Concurrent Version System[1]) and RCS (Revision Control System[2]).

[1] http://ximbiot.com/cvs
[2] http://www.gnu.org/software/rcs/rcs.html

Nevertheless, a distinguished functionality of the version concept appears when applied to structured data and managed by database systems. In this context, it is possible to control the evolution of different elements of the data depending on the respective data model (e.g. relations, tuples, columns for relational model and classes, attributes, relationships, methods for object oriented model). Similarly, versioning techniques have also appeared in a large range of topics. Some recent examples are XML document management [Vagena et al. 2004], semantic web [Noy and Musen 2004], and data warehousing [Wrembel and Morzy 2005].

Modern database systems provide not only powerful data models but also complex query languages for distinguished features such as the ability to handle object versions. In this sense query languages have evolved to include several features from complex programming languages. This situation has lead to incrementing their definition with techniques and methodologies largely used by the programming language community, such as formal semantics and type systems.

ODMG has striven to integrate object oriented databases capabilities with object oriented programming languages [Cattell et al. 2000]. A type system for an OQL-like query language is formally specified in [Bierman and Trigoni 2000, Bierman 2003], and a formal semantics for OQL is defined in terms of an object algebra in [Zamulin 2003]. Two complementary concerns are to provide a formal semantics for XML related query languages such as XPath, XQuery [Draper et al. 2005, Colazzo et al. 2002, Siméon and Wadler 2003] and XDuce [Hosoya and Pierce 2000], and to integrate XML and SQL by using domain specific embedded languages [Christensen et al. 2002, Graunke et al. 2001, Kiselyov and Krishnamurthi 2003, Thiemann 2002, Welsh et al. 2002, Bierman et al. 2005].

In this work we formally define a query language for a non-conventional database that supports the concept of object versions. Information stored in a database evolves with time and, very often, it is necessary to maintain and to retrieve versions that keep track of such evolution. This new language, named *Versioned Object Query Language* (VOQL), extends ODMG [Cattell et al. 2000] with distinguished features for recovering object versions. We present VOQL's main versioning features through a series of examples and we discuss the main aspects of an interpreter available for VOQL.

Similarly to the core aspects of its counterpart OQL (formally defined in [Bierman and Trigoni 2000, Bierman 2003]), we also provide a precise definition of VOQL in the form of a type system and we prove it safe in relation to a small-step operational semantics. This formal definition serves two purposes: it is essential for proving that the language is *safe* (in the sense that evaluation of well-typed queries do not lead to execution errors) and it also works as a precise language reference for guiding an implementation process.

The rest of this paper is structured as follows. Section 2 presents the *Versioned Obect Data Model* (VODM) on which VOQL is based. Section 3 presents the VOQL query language through a series of examples and illustrates how it can be used to recovery versioned objects and their versions. Section 4 introduces an operational semantics for VOQL and Section 5 defines a type system for the language. Section 6 proves that the type system is safe with respect to the operational semantics. Section 7 presents an interpreter for VOQL. Section 8 goes over related work and section 9 concludes this paper.

## 2 Versioned Data Model

In this section we define the *Versioned Object Data Model* (VODM), a class-based object model based on ODMG [Cattell et al. 2000] and extended with new features for specifying object versions.

A VODM database schema is a collection of class definitions similar to classes defined in Java. A class defines a set of *attributes*, *relationships*, and *method signatures*. A relationship is defined explicitly. Transversal paths are declared in pairs, one for each direction of the relationship. Relationships are accessed by the query language as attributes (i.e. from the semantics point of view, attributes and relationships receive the same treatment). The database management system is responsible for maintaining the referential integrity of relationships. So, if an object that participates in a relationship is deleted, any transversal path to that object must also be deleted. Maintaining referential integrity ensures that applications cannot dereference transversal paths that lead to nonexistent objects. Finally, the class behavior is specified as a set of method signatures. In ODMG, methods are defined in a host programming languages such as Java and C♯. Hence, we do not consider methods in our formal treatment.

Once the database is active, the objects are instantiated and versions can be created implicitly (any update defines a new version) or explicitly (by a user). For each set of versions, there is a *versioned object* in charge of grouping all versions of the same object. Each versioned object has also a *current version* which, by default, is the most recently created. Again, it is not the focus of this paper to discuss how exactly this whole process works. What matters is that the database is capable of managing different versions for the same object following the basic features aforementioned.

For instance, consider an example taken from an academic system. The curricula usually undergoes successive revisions, such as an update on the set of courses to be taken in order to obtain a degree. Students must follow the current curriculum, but it might be necessary to query past curriculum versions in order to grant course equivalences for instance. Figure 1 illustrates the graphical representation of the extension of the class Curriculum. This extension has one

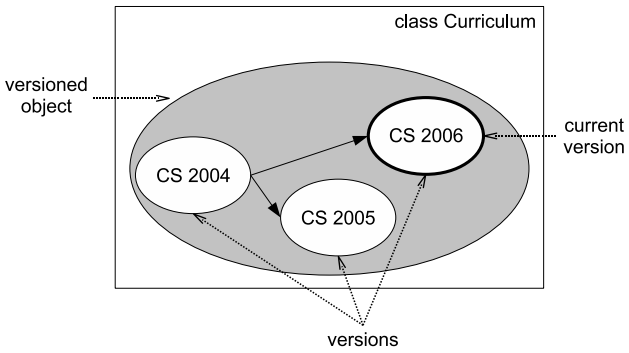



Figure 1: Graphical representation of a versioned object of the class Curriculum with three versions (CS 2004, CS 2005 and CS 2006).

versioned object with three versions of a Computer Science curriculum: the first one - CS2004, and two derived alternatives - CS 2005 and CS 2006.

Figure 2 shows a fragment of a VODM schema for an academic system structure. There are three classes named `Person`, `Faculty`, and `Course`. In this fragment, class `Person` has attributes for last name and social security number; `Faculty` extends `Person` with office number and title; `Course` has number, description, and units. There is a relationship (with transversals called `teaches` and `is_taught_by`) between classes `Faculty` and `Course`. As illustrated, the main clauses on a VODM specification are as follows:

- `extends`: defines a single inheritance relationship between two classes. For simplicity, we assume that all class definitions explicitly state a superclass and the class `Object` is the superclass of all classes.
- `extent`: names the set of all instances of a class within a particular database state. In the example, the extent `FacultyM` is a subset of the extent `Persons`, as `Faculty` is a subclass of `Person`.
- `hasVersions`: allows storing different versions of objects, the main feature of VODM. This clause (in a class definition) indicates that modifications in the state of an object of the class leads to the creation of a version of that object. Instances of a versioned class are called *versioned objects*.

Observe that the versioned object's versions are organized in a version derivation tree. A language for querying a database, where instances are organized in such a way, should provide support for collecting all versions of a versioned object, and for navigating among the nodes of the derivation tree (e.g. selecting the original and the last versions, and selecting the antecessors/successors of a

```
class hasVersion Course extends Object
extent Courses
{   attr courseNum : String,
    attr  courseDesc : String,
    rel  is_taught_by : Professor (inverse Professor teaches)
}

class hasVersion Person extends Object
extent Persons
{   attr lastName :String,
    attr SSN : Integer,
}

class hasVersion Faculty extends Person
extent FacultyM
{  attr officeN : String,
   attr title : String,
   rel teaches : Set(Courses)(inverse Course is_taught_by)
}
```

**Figure 2:** VODM schema for a fragment of an academic system.

specific version). In the next section we present the VOQL language informally through a series of examples.

## 3 VOQL Query Language

Here, we propose the *Versioned Object Query Language* (VOQL) as a query language for this database with versioning support. This language is a fragment of ODMG OQL [Cattell et al. 2000], extended with features for maintaining object versions. We consider only functional aspects of the query language, i.e, queries that do not create and/or delete objects.

VOQL queries belong to the language defined by the grammar of Figure 3. Next, we focus on the new features presented by VOQL.

### 3.1 Extended features

VOQL has variables of types boolean, integer, strings, and sets. When a query $q$ denotes a versioned object, $q.a$ accesses the attribute $a$ of the versioned object's

| | | |
|---|---|---|
| $q \in$ *Query* | | |
| $q ::=$ | $b$ | booleans |
| | $i$ | integers |
| | $s$ | strings |
| | $\{q_1, \ldots, q_n\}$ | sets, $n \geq 0$ |
| | *vo_id* | versioned object ids |
| | *v_id* | version ids |
| | $x$ | variables |
| | $q_1$ *bop* $q_2$ | binary operations |
| | *uop* $q_1$ | unary operations |
| | $q.a$ | attribute access |
| | $q$`->`$a$ | attribute access in a colection |
| | $q$`->versions` | acess to all versions |
| | `if` $q_1$ `then` $q_2$ `else` $q_3$ | conditional |
| | `select` $q_a$ `from` $q_1\ x_1, \ldots, q_n\ x_n$ `where` $q_b$ | selection, $n \geq 0$ |

**Figure 3:** VOQL syntax.

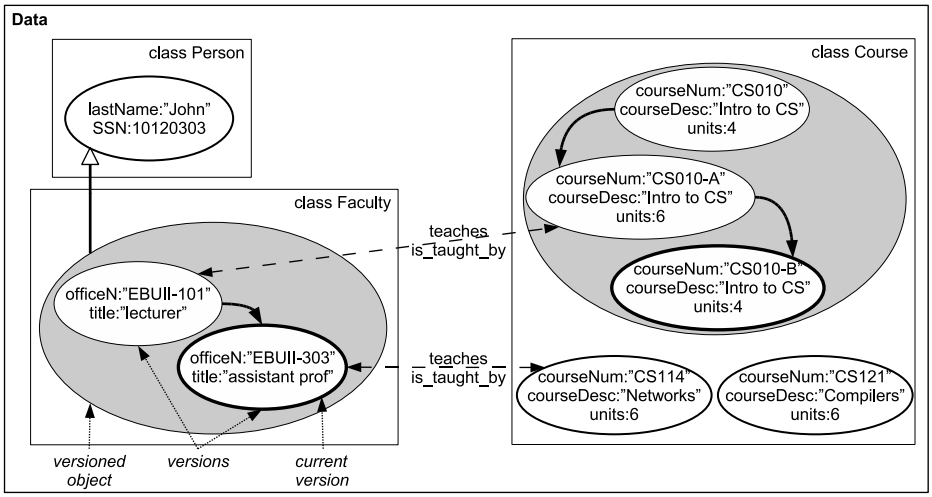
current version. Additionally, in the notation $q$`->`$a$, $q$ is a set of versioned objects or versions. The result of $q$`->`$a$ is a collection of all $a$-attributes of elements of $q$.

Besides the regular arithmetic, logical and relational operators, VOQL has unary operators that test the absolute position of a version (`is_leaf`, `is_root`) and binary operators that compare the relative position of two versions in the version derivation tree of a versioned object (`is_succ`, `is_pred`). VOQL has the usual conditional and `select-from-where` constructs as well.

### 3.2 Basic and versioned queries

In this subsection we informally describe the principles of VOQL with emphasis on its versioning capabilities. We consider again an academic system (as specified in Figure 2) with a fragment of its instances illustrated on the top half of Figure 4. Class `Person` has only one instance. Class `Faculty` has one versioned object with two versions. Finally, class `Course` has three different instances: Networks and Compilers with one instance each, and Introduction to Computer Science with three versions. The bottom half of Figure 4 has a series of examples of VOQL queries. In the figure, each query has a proper identifier that will be used in the text.

Query *1* is a plain (with no versions) query that retrieves the set of `titles` of all faculty members named "`John`". The variable `FacultyM` represents the extension of the class `Faculty`. The names of extents (previously declared within the database schema) are treated as global variables within queries. On the other hand, the variable `f` is local to the query. Since no specific version is required




| | | |
|---|---|---|
| **Query 1**<br>select f.title<br>from FacultyM f<br>where f.lastName = "John"<br>**Result**<br>title: "assistant prof" | **Query 2**<br>select f.title<br>from FacultyM-> versions f<br>where f.lastName = "John"<br>**Result**<br>title: "lecturer", "assistant prof" | **Query 3**<br>select c.courseNum<br>from Courses->versions c<br>where (is_root c)<br>**Result**<br>courseNum: "CS010", "CS114", "CS121" |
| **Query 4**<br>select c.courseNum<br>from Courses c<br>where c.units > 4<br>**Result**<br>courseNum: "cs114", "cs121" | **Query 5**<br>select c.courseNum<br>from FacultyM.teaches c<br>where c.units > 4<br>**Result**<br>courseNum: "cs114" | **Query 6**<br>select c.courseNum<br>from FacultyM->teaches c<br>where c.units > 4<br>**Result**<br>courseNum: "cs114" |

Figure 4: Graphical representation of some versions from the academic system plus queries and their results.

in this query, `f` ranges over the current version in the extent `FacultyM`. Adding the word `versions` to the extent name changes the query range to consider all its versions. Therefore, query *2* retrieves the collection of titles of all versions of faculty members named "`John`".

Suppose now that one wants to retrieve the original course numbers of all course instances. In VOQL this is achieved by query *3* that uses the operator `is_root` for testing the position of versions within the version derivation tree. This query returns a collection of course numbers only from those courses's versions that are root on the derivation tree of the versioned object. Note that courses *CS114* and *CS121* do not have any version. They belong to the results of this query because the first instance of an object is regarded as its first version and root of the derivation tree, by default.

### 3.3 The '.' operator

In OQL, the '.' operator has a position-dependent semantics. When used in the statements `select` and `where` of a query, it implies conventional member access. For example, in query *4*, the operator is applied directly to the variable `c`, which ranges over object identifiers. This query returns the set of `courseNum` of courses with more than 4 units. On the other hand, when used in the statement `from`, it expresses the creation of a collection of attributes from a collection of object identifiers. For example, in query *5*, the '.' operator is used in conjunction with the extent `FacultyM`. It generates a collection by reaching the `teaches` relationship of all elements within the extent and grouping them all. The result is different from query *4* because *5* only considers courses being taught (i.e. course *CS121* is not returned since it has no faculty assigned to).

In order to have a simpler context-independent, non-positional semantics, we distinguish these two different meanings with two operators: '.' for member access and $\rightarrow$ for iteration over collections. For example, query *5* is then formulated as query *6* in VOQL. The ambiguity in the interpretation of the '.' operation (not mentioned in ODMG OQL) has been firstly pointed out in [Galante et al. 2003a]. This distinction became clear while we were working on the formal semantics and the type system for VOQL.

## 4 Operational Semantics for VOQL

The result of query evaluation depends on the *extent*, *versioned object* and *version* environments. We first explain in detail each one of these environments, then we discuss the semantic rules.

### 4.1 Environments

From now on, we assume the existence of sets of class, extent and attribute identifiers (*ClassId, ExtId*, and *AttrId* respectively).

**Extent environment**. An *extent* is a collection of versioned object identifiers (*vo_id*) that correspond to all persistent versioned objects of a given class. An *extent environment* is a mapping from extent identifiers (defined in the schema with the `extents` clause) to extents. We use *vee* as a metavariable for extent environments:

$$vee \in VEE = ExtId \xrightarrow{fin} \mathcal{P}(Vo\_id)$$

**Versioned object environment**. The state of a versioned object (*vobjstate*) is given by a tuple (*c, versions, current, deriv)* where *c* is the versioned object's

class name, *versions* is a set with the identifiers of all the versions of this versioned object, *current* is the identifier of the current version of the versioned object, and *deriv* is a versioning relation on elements of *versions*.

$$vobjstate \in Vobjstate = ClassId \times \mathcal{P}(V\_id) \times V\_id \times \mathcal{P}(V\_id \times V\_id)$$

For instance, The versioning relation for the versioned object of figure 1 is given by the set { (`CS 2004`, `CS 2005`), (`CS 2004`, `CS 2006`)}.

A *versioned object environment* maps each versioned object identifier *vo_id* to a pair *(c, vobjstate)* where *c* and *vobjstate* are the versioned object's class name and state respectively. We use *voe* as a metavariable for versioned object environments.

$$voe \in VOE = Vo\_id \xrightarrow{fin} ClassId \times Vobjstate$$

**Version environment**. In semantics for object oriented languages the state of an object maps object's attributes to their *values*. Here the attribute of a version can hold not only values but also versioned object identifiers. Versioned object identifiers are not considered as values because, depending on the context, they can still be evaluated to the versioned object's *current* version. The content of an attribute (ranged over by the metavariable $k$) is then an element of the set *Stored* defined as:

| | | |
|---|---|---|
| $k \in$ | $Stored$ | |
| $k ::=$ | $b$ | booleans |
| $\mid$ | $i$ | integers |
| $\mid$ | $s$ | strings |
| $\mid$ | $vo\_id$ | versioned object ids |
| $\mid$ | $v\_id$ | versions ids |
| $\mid$ | $\{k_1, \ldots, k_n\}$ | sets $n \geq 0$ |

The state of a version is then given by a function that maps the names of the version's attributes to their *stored contents*:

$$vstate \in VersionState = AttrId \rightarrow Stored$$

A *version environment* maps version identifiers to pairs *(c, vstate)* where *c* and *vstate* are the version's class name and state respectively. We use *ve* as a metavariable for version environments.

$$ve \in VE = V\_id \xrightarrow{fin} ClassId \times VersionState$$

Moreover, a database state is represented by a triple $(vee, voe, ve)$ of extent, versioned object and version environments respectively:

$$(vee, voe, ve) \in VDBState = VEE \times VOE \times VE$$

We write $vee; voe; ve \vdash q \ \rightarrow \ q'$ to represent that the VOQL query $q$ is reduced, in one step, to the VOQL query $q'$ in database state $(vee; voe; ve)$. Finally, the following subset of queries cannot be further reduced:

$$\begin{array}{lll} v \in Values & & \\ v ::= b & & \textit{booleans} \\ \quad | \ i & & \textit{integers} \\ \quad | \ s & & \textit{strings} \\ \quad | \ v\_id & & \textit{version ids} \\ \quad | \ \{v_1, \ldots, v_n\} & \textit{set of values} & n \geq 0 \end{array}$$

In order to reduce the number of rules we use *evaluation contexts*. An evaluation context $\mathcal{E}$ is a query with a single hole in it (written $\bullet$) that marks the position of the next subexpression to be reduced. The following grammar defines evaluation contexts for VOQL:

$$\begin{array}{l} \mathcal{E} ::= \bullet \\ \quad | \ \{\mathbf{v}, \mathcal{E}, \mathbf{q}\} \\ \quad | \ \mathcal{E} \ bop \ q \\ \quad | \ v \ bop \ \mathcal{E} \\ \quad | \ uop \ \mathcal{E} \\ \quad | \ \mathcal{E}.a \\ \quad | \ \mathcal{E}\texttt{->}a \\ \quad | \ \texttt{if} \ \mathcal{E} \ \texttt{then} \ q_1 \ \texttt{else} \ q_2 \\ \quad | \ \texttt{select} \ q_a \ \texttt{from} \ \mathcal{E} \ x_1, \ldots, q_n \ x_n \ \texttt{where} \ q_b \\ \quad | \ \texttt{select} \ q \ \texttt{from where} \ \mathcal{E} \\ \quad | \ \mathcal{E}\texttt{->versions} \end{array}$$

According to this grammar, sets of queries are evaluated left-to-right. Also, in a *select-from-where*, the *from* part is evaluated first followed by the evaluation of the *where* part. A key property is that a query is either a value or uniquely decomposed as an evaluation context with a hole filled with a non-value subexpression that matches the query in the left side of a reduction rule. The rule Ctx connects the evaluation of a non-value subexpression with its evaluation context:

$$\frac{vee; voe; ve \vdash q \ \rightarrow \ q'}{vee; voe; ve \vdash \mathcal{E}[q] \ \rightarrow \ \mathcal{E}[q']} \qquad (\textsc{Ctx})$$

### 4.2 Evaluation Rules

We know proceed by showing the evaluation rules of queries filling the holes in evaluation contexts.

**Basic Queries.** The `select-from-where` construct is evaluated by first building the cartesian product of all extents mentioned in the `from` statement. From this product, only those objects that satisfy the condition specified within `where` are collected. The result of the query is given by the `select` statement that projects components of these objects.

If the first query in the `from` part is the empty set, the whole query reduces to the empty set by the rule SELECT1.

$$vee; voe; ve \vdash \texttt{select}\ q_a\ \texttt{from}\ \{\}\ x_1, \ldots, q_m\ x_n\ \texttt{where}\ q_b \ \rightarrow\ \{\}\ (\textsc{Select1})$$

The cartesian product is given by SELECT2 that uses the substitution operation [3] to produce all possible combinations of values for all variables within the `from` statement. Note that the query in the right side of SELECT2 forms another query using the `union` set operator.

$$\begin{array}{c} vee; voe; ve \vdash \texttt{select}\ q_a\ \texttt{from}\ \{v_1, \ldots, v_m\}\ x_1, \ldots, q_n\ x_n\ \texttt{where}\ q_b \ \rightarrow \\ (\texttt{select}\ q_a\ \texttt{from}\ q_2\ x_2, \ldots, q_n\ x_n\ \texttt{where}\ q_b)[x_1 ::= v_1] \\ \texttt{union} \\ \ldots \\ \texttt{union} \\ (\texttt{select}\ q_a\ \texttt{from}\ q_2\ x_2, \ldots, q_n\ x_n\ \texttt{where}\ q_b)[x_1 ::= v_m] \end{array} \quad (\textsc{Select2})$$

When the `where` part is `false`, the result of the `select` is the empty set. Otherwise, the whole `select` proceeds with a set with $q_a$.

$$vee; voe; ve \vdash \texttt{select}\ q_a\ \texttt{from where false} \ \rightarrow\ \{\} \quad (\textsc{Select3})$$

$$vee; voe; ve \vdash \texttt{select}\ q_a\ \texttt{from where true} \ \rightarrow\ \{q_a\} \quad (\textsc{Select4})$$

**Versioned Objects.** Rule VAR reduces an extent identifier to its associated collection of versioned object identifiers.

$$\frac{vee(x) = \{vo\_id_1, \ldots vo\_id_2\}}{vee; voe; ve \vdash x \ \rightarrow\ \{vo\_id_1, \ldots vo\_id_2\}} \quad (\textsc{Var})$$

The rule VO_ID evaluates a reference to a versioned object to its current version according to the versioned object environment.

[3] The notation $q[x := v]$ denotes the query that results from the substitution of $v$ for all free occurrences of $x$ in $q$

$$\frac{voe(vo\_id) = (c, versions, current, tree)}{vee; voe; ve \vdash vo\_id \;\; \rightarrow \;\; current} \quad (\textsc{Vo\_id})$$

**Operators.** A version derivation tree of a versioned object is a set of pairs of version identifiers. If a pair $(v\_id_1, v\_id_2)$ belongs to this set, it means that the version $v\_id_1$ is an immediate predecessor of version $v\_id_2$. Next, the premises of rules Pred1 and Pred2 check whether $v\_id_1$ is predecessor of $v\_id_2$ in the version derivation tree.

$$\frac{voe(vo\_id) = (c, versions, current, tree) \qquad (v\_id_1, v\_id_2) \in tree}{vee; voe; ve \vdash v\_id_1 \texttt{ is\_pred } v\_id_2 \;\; \rightarrow \;\; \texttt{true}} \quad (\textsc{Pred1})$$

$$\frac{voe(vo\_id) = (c, versions, current, tree) \qquad (v\_id_1, v\_id_2) \notin tree}{vee; voe; ve \vdash v\_id_1 \texttt{ is\_pred } v\_id_2 \;\; \rightarrow \;\; \texttt{false}} \quad (\textsc{Pred2})$$

The rules for the other positional operators follow this same pattern and, hence, are omitted (rules for arithmetic, logical and relational operators are also omitted).

**The '.' operator.** As previously discussed, the ambiguity of the '.' operator is solved by presenting two operators with distinguished meanings. The rule Dot1 accesses the content of an object attribute.

$$\frac{ve(v\_id) = (c, vstate) \qquad vstate(a) = k}{vee; voe; ve \vdash v\_id.a \;\; \rightarrow \;\; k} \quad (\textsc{Dot1})$$

Rules Arrow1 and Arrow2 evaluate queries of the form $\{v\_id_1, \ldots v\_id_n\}\texttt{->}a$ $(n \geq 0)$.

$$vee; voe; ve \vdash \{v\_id, \mathbf{v}\}\texttt{->}a \;\; \rightarrow \;\; \{v\_id.a\} \texttt{ union } \{\mathbf{v}\}\texttt{->}a \quad (\textsc{Arrow1})$$

$$vee; voe; ve \vdash \{\}\texttt{->}a \;\; \rightarrow \;\; \{\} \quad (\textsc{Arrow2})$$

Finally, rule Versions collects all versions of a versioned object by accessing the versioned object state.

$$\frac{(c_i, versions_i, current_i, tree_i) \in Image(voe) \qquad v\_id_i \in versions_i}{vee; voe; ve \vdash \{v\_id_i\}\texttt{->versions} \;\; \rightarrow \;\; \bigcup_{i=1}^{n} versions_i} \quad (\textsc{Versions})$$

## 5 Type System

Types in VOQL are given by the following grammar:

$$\begin{aligned} \sigma \in\ & \mathit{Type} \\ \sigma ::= & \ \texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid c \mid \texttt{set}(\sigma) \end{aligned}$$

A type judgement for a VOQL query $q$ has the form

$$\mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma \vdash q : \sigma$$

where *vee, voe* and *ve* are extent, versioned object and version environments respectively, *vsch* is a representation of a database schema, and $\Gamma$ is an environment mapping query variables to their types.

A schema is represented by a mapping from class identifiers to class definitions. A class definition in its turn, is given by a triple $(s, e, ty)$ where the first component $s$ is the name of the superclass, the second component $e$ is the name of the class extent and the final component $ty$ is a map from the class attributes to their types. For clarity, a typing context *vsch;vee;voe;ve;*$\Gamma$ is sometimes represented by $K$.

**Basic queries.** Rule VAR is for typing both local and global query variables (extent identifiers as defined in the schema):

$$\frac{\Gamma(x) = \sigma}{\mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma \vdash x : \sigma} \quad \text{(VAR)}$$

Rule T-SELECT types the queries $q_1 \ldots q_n$ in the `from` part from left to right adding to the type context $\Gamma$ the associated type assignments to variables $x_1 \ldots x_n$.

$$\frac{\begin{array}{r} \mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma \vdash q_1 : \texttt{set}(\sigma_1) \\ \mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma, x_1 : \sigma_1 \vdash q_2 : \texttt{set}(\sigma_2) \\ \vdots \\ \mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma, x_1 : \sigma_1, \ldots, x_{n-1} : \sigma_{n-1} \vdash q_n : \texttt{set}(\sigma_n) \\ \mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash q_a : \sigma_a \\ \mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash q_b : \texttt{bool} \\ x_1 \text{ to } x_n \text{ different} \qquad x_1, \ldots, x_n \notin \mathit{Dom}(\Gamma) \qquad n \geq 0 \end{array}}{\mathit{vsch}; \mathit{vee}; \mathit{voe}; \mathit{ve}; \Gamma \vdash \texttt{select}\ q_a\ \texttt{from}\ q_1\ x_1\ , \ldots, q_n\ x_n\ \texttt{where}\ q_b : \texttt{set}(\sigma_a)} \quad \text{(T-SELECT)}$$

**Versioned objects.** Rules T-VOID and T-VID access the versioned object environment and the version environment to obtain the class name.

$$\frac{voe(void) = (c, vobjstate)}{K \vdash void : c} \quad \text{(T-Void)}$$

$$\frac{ve(vid) = (c, vstate)}{K \vdash vid : c} \quad \text{(T-Vid)}$$

**Operators.** Rule T-Pred simply requires that both operands of `is_pred` be of the same class type $c$. The rules for other positional operators are similar:

$$\frac{K \vdash q_1 : c \qquad K \vdash q_2 : c}{K \vdash q_1 \texttt{ is\_pred } q_2 : \texttt{bool}} \quad \text{(T-Pred)}$$

**The '.' operator.** Rule T-Dot states that if $q$ is of type $c$, and if the type of attribute $a$ is $\sigma$ based on the schema environment for $c$, then $q.a$ is of type $\sigma$.

$$\frac{K \vdash q : c \qquad vsch(c) = (s, e, ty) \qquad ty(a) = \sigma}{K \vdash q.a : \sigma} \quad \text{(T-Dot)}$$

Observe that the type of a query $q$`->`$a$, given by rule T-Arrow, is $\texttt{set}(\sigma)$, where $a$ is of type $\sigma$ according to the schema.

$$\frac{K \vdash q : \texttt{set}(c) \qquad vsch(c) = (s, e, ty) \qquad ty(a) = \sigma}{K \vdash q\texttt{->}a : \texttt{set}(\sigma)} \quad \text{(T-Arrow)}$$

The rule T-Versions ensures that the construction `versions` will be applied only to collections of versioned objects.

$$\frac{K \vdash q : \texttt{set}(c)}{K \vdash q\texttt{->versions} : \texttt{set}(c)} \quad \text{(T-Versions)}$$

**Subtyping.** The subtyping relation for VOQL types, written $\sigma \leq \sigma'$, is the reflexive and transitive closure of the relation defined by the next rules. We assume that the subclass hierarchy specified in the schema definition is represented by a relation written **extends**:

$$\frac{c \ \textbf{ extends } \ c'}{c <: c'} \quad \text{(Extends)}$$

$$\frac{\sigma_1 <: \sigma_2}{\texttt{set}(\sigma_1) <: \texttt{set}(\sigma_2)} \quad (<:\text{Set})$$

Rule T-Sub connects the subsumption relation $\leq$ with the type system.

$$\frac{K \vdash q : \sigma_1 \qquad \sigma_1 <: \sigma_2}{K \vdash q : \sigma_2} \quad \text{(T-Sub)}$$

## 6 Safety of VOQL type system

In order to prove type safety the three environments that form the database state must be valid and must also be well-formed according to the database schema. For instance, a versioned identifier cannot belong to more than one versioned object, and the type specified for an attribute in the schema must match the type of the stored content of this attribute. The relation which states that database environments are valid and are well-formed according to the database schema is written

$$vsch \vdash vee, voe, ve$$

The definition of this relation is straightforward but too lengthy, so it is not presented in this paper.

In VOQL, extent identifiers are lexically indistinguishable from variables introduced in the `from` part of a `select-from-where`. This generates a problem to the type system, which needs to know when certain identifiers refer to an extent. A solution is to start with an initial type environment $\Gamma$ containing type associations for the extent identifiers extracted from the schema.

The following two theorems state that, when the database state is well formed according to the database schema, the reduction of well typed VOQL queries do not get stuck, and that the types of well-typed VOQL queries are preserved by the reduction rules as well.

**Theorem 1 (Progress)** *If*

- $vsch; vee; voe; ve; \Gamma \vdash q : \sigma$, *and*
- $vsch \vdash vee, voe, ve$,

*then either*

- $q \in Values$, *or*
- *there is* $q'$ *such that* $vee; voe; ve \vdash q \rightarrow q'$.

**Theorem 2 (Type Preservation)** *If*

- $vsch; vee; voe; ve; \Gamma \vdash q : \sigma$,
- $vee; voe; ve \vdash q \rightarrow q'$, *and*
- $vsch \vdash vee, voe, ve$,

*then* $vsch; vee; voe; ve; \Gamma \vdash q' : \sigma$.

The proofs of the two theorems above are straightforward inductions on the structure of queries and for this reason are omitted from the paper. They follow the classical syntactical soundness approach for type soundness of [Wright and Felleisen 1994].

## 7 VOQL Interpreter

This section describes the VOQL interpreter developed for experimenting with the language. The VOQL Interpreter consists of three main modules, as depicted in Figure 5: the *query evaluator*, the *type checker*, and the *integrity checker*. The query evaluator and the type checker were implemented following strictly the operational and typing rules for the language. Although the DBMS is responsible for the database integrity, we decided to provide a verifier of valid database states (the integrity checker module) in relation to the schema since it is required for query evaluation safety.

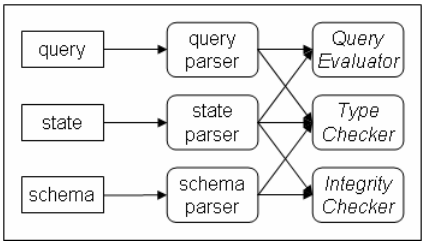



**Figure 5:** Architecture of VOQL implementation.

Figure 6 illustrates a screenshot of the VOQL Interpreter. Queries can be specified through a graphical interface that displays query types and results. The interface also allows users to define database schema and extents. The input data for the VOQL Interpreter are:

1. *query* to be evaluated;
2. *database state* describing extents and their objects; and
3. *database schema*, defining the classes of objects.

All these elements are described textually, and parsers construct their internal representation.

The interpreter was implemented in OCaml [OCaml 2006], using the `ocamllex` and `ocamlyacc` tools for building the needed parsers. The graphical user interface was built using the `labltk` library. The VOQL interpreter runs on all platforms supported by the OCaml interpreter, including MS-Windows and GNU/Linux. Source code, binaries and documentation can be found in `http://www.inf.ufrgs.br/~rma/ctvql`.

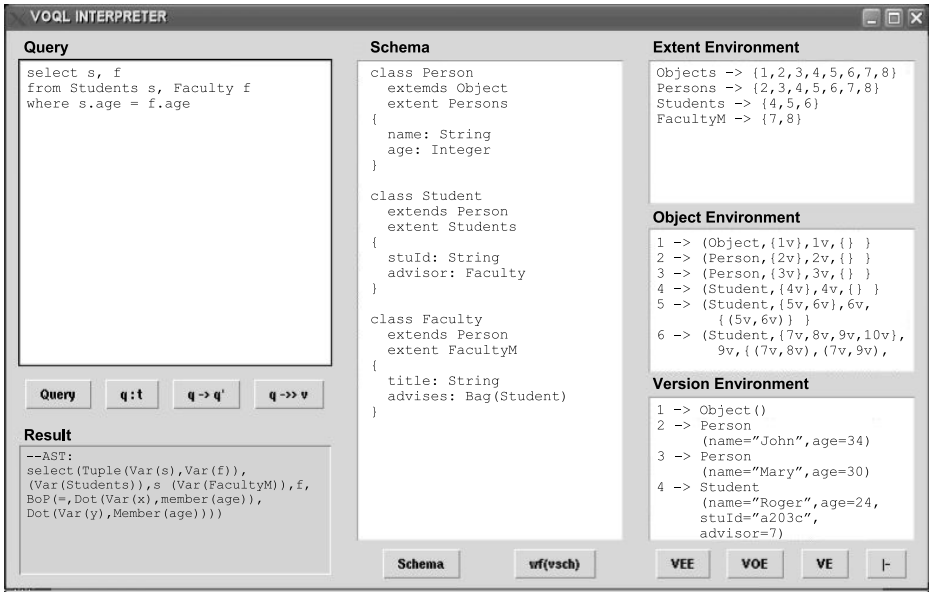



**Figure 6:** Screenshot of VOQL implementation.

## 8 Related Work

The concept of versioning has been used in many different applications over the years. A distinguished functionality of the concept appears when data versioning is managed by database systems. In such scenario, the *Temporal Versions Model* (TVM) [Moro et al. 2001b, Moro et al. 2001a], was proposed as a join solution for managing both temporal and versioned data. The query language for this model, named *Temporal Versioned Query Language* (TVQL), was proposed in [Moro et al. 2002]. Likewise, there was also a preliminary work on extending ODMG and OQL with both temporal and versioning features, named TV_ODMG [Gelatti et al. 2002].

In order to propose VOQL, we assumed the existence of a database with versioning support. The base model has some common characteristics to both TVM and TV_ODMG. Specifically, the features for handling versions, i.e. the semantics of versioned object and versions, are mostly the same. Nevertheless, we opted for not including the temporal aspects in our work because the versioning management provided by a database system is orthogonal to the temporal one (as the concepts are managed independently from each other). Also, our proposal is more comprehensive since it covers some important features overseen by previous work, for example the distinction between . and $\rightarrow$ operators. More importantly,

there is no formal specification, operational semantics or type system specified for the TVM query language or its TV ODMG counterpart. The idea of covering the temporal aspects with versioning is left as future work.

When databases have complex structures that require constant changes, the schema may also need to be versioned. In this scenario, models and techniques for schema versioning are proposed for managing the undergoing new specifications and user requirements [Galante et al. 2005]. Moreover, [Galante et al. 2003a] proposes a language for controlling the schema evolution process, named *Temporal and Versioning Language for Schema Evolution* (TVL/SE). This language is able to derive and modify schema versions, and also to update data associated with them, creating either new object versions or just keeping the history of these data modifications. A formal semantics for TVL/SE is given in [Galante et al. 2005, Galante et al. 2003b]. Our work is complementary to those since VOQL is specific for the data level, as opposed to the schema level.

Also related to our work is the type system formally specified for a complex-value OQL-like query language in [Bierman and Trigoni 2000, Bierman 2003]. An operational semantics for query evaluation is also given and this semantics is used to prove the soundness of the proposed type system. In [Zamulin 2003] a formal semantic for OQL is defined in terms of an object algebra and OQL queries are translated into corresponding object algebra expression. None of these works, though, has a formal definition for a query language with versioning support.

Our work follows the same lines of a series of recent research developments unifying the areas of database and the formal definition of programming languages. Besides the related work on object oriented database languages already mentioned, there is also an intense activity with semi-structured databases, specially with XML related languages such as XPath and XQuery [Draper et al. 2005]. A type system for a XML query language is given in [Colazzo et al. 2002] and is used to verify whether the query language operations respect the XML schema restrictions. [Siméon and Wadler 2003] specifies a formal semantic for XML Schema while W3C Consortium [W3C XML Work Group 2006] has an effort to specify an operational semantics and a type system for both XPath and XQuery [Draper et al. 2005].

## 9 Summary and Future Work

In this work, we formally proposed a query language – the Versioned Object Query Language (VOQL) – for extending ODMG Object Query Language (OQL) with new features to recover object versions. VOQL has a set of operators and constructs that make it suitable for querying a database where versions are organized in a derivation tree. We also defined a type system for VOQL and we

proved it safe in relation to a small-step operational semantics. Furthermore, we solved the ambiguity of the '.' operator first identified in [Galante et al. 2003a]. We also described a VOQL interpreter developed for validating the proposed formal definitions.

Finally, we acknowledged the importance of a database model that is capable of managing time aspects besides the versioning support. An initial extension to ODMG for such a model is presented in [Gelatti et al. 2002]. Since there is no formal specification of the language for that model, we plan to extend VOQL for time support as well.

## References

[Bierman 2003] Bierman, G. M. (2003). Formal semantics and analysis of object queries. In *ACM SIGMOD International Conference on Management of Data*, pages 407–418, San Diego, California, USA. New York: ACM Press.

[Bierman et al. 2005] Bierman, G. M., Meijer, E., and Schulte, W. (2005). The essence of data access in C$\omega$. In *Proceedings of ECOOP*, volume 2736 of *LNCS*, pages 287–311. Springer-Verlag.

[Bierman and Trigoni 2000] Bierman, G. M. and Trigoni, A. (2000). Towards a formal type system for ODMG OQL. Technical report, University of Cambridge.

[Cattell et al. 2000] Cattell, R., Barry, D., Bartels, D., Berler, M., Eastman, J., Gamerman, S., Jordan, D., Springer, A., Strickland, H., and Wade, D. (2000). *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco. 280p.

[Christensen et al. 2002] Christensen, A. S., Muller, A., and Schwartzbach, M. I. (2002). Static analysis for dynamic XML. In *Proceedings of PlanX*.

[Colazzo et al. 2002] Colazzo, D., Ghelli, G., Manghi, P., and Sartiani, C. (2002). Types for correctness of queries over semistructured data. In *WebDB*, pages 19–24.

[Conradi and Westfechtel 1998] Conradi, R. and Westfechtel, B. (1998). Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282.

[Draper et al. 2005] Draper, D., Fankhauser, P., FERNNDEZ, M., Malhota, A., Rose, K., Rys, M., SIMON, J., and Wadler, P. (2005). XQuery 1.0 and XPath 2.0 formal semantics. In W3C Working Draft. ¡http://www.w3.org/TR/2003/WD-xquery-semantics-20030502/¿.

[Galante et al. 2005] Galante, R. M., dos Santos, C. S., Edelweiss, N., and Álvaro Freitas Moreira (2005). Temporal and versioning approach to schema evolution in object-oriented databases. *Data & Knowledge Engineering*, 53(2):99–128.

[Galante et al. 2003a] Galante, R. M., Edelweiss, N., and dos Santos, C. S. (2003a). TVL_SE: Temporal and Versioning Language for Schema Evolution in Object-Oriented Databases. In *Intl. Conf. on Database and Expert Systems Applications*, volume 2736 of *LNCS*, pages 683–692. Berlin: Springer-Verlag.

[Galante et al. 2003b] Galante, R. M., Edelweiss, N., dos Santos, C. S., and Álvaro Freitas Moreira (2003b). Data modification language for full support of temporal schema versioning. In *Brazilian Symposium on Databases*, pages 114–128. UFAM.

[Gelatti et al. 2002] Gelatti, P. C., dos Santos, C. S., and Edelweiss, N. (2002). TV_ODMG: Uma Extensão do Padrão ODMG com Suporte para Tempo e Versões (in Portuguese). In *SBBD*, pages 42–56.

[Graunke et al. 2001] Graunke, P., Krishnamurthi, S., Hoeven, S. V. D., and Felleisen, M. (2001). Programming the web with high-level programming languages. In *Proceedings of ASE*.

[Hosoya and Pierce 2000] Hosoya, H. and Pierce, B. C. (2000). Xduce: A typed xml processing language (preliminary report). In *WebDB (Selected Papers)*, pages 226–244.

[Katz 1990] Katz, R. H. (1990). Towards a unified framework for version modeling in engineering databases. *ACM Computing Surveys*, 22(4):375–408.

[Kiselyov and Krishnamurthi 2003] Kiselyov, O. and Krishnamurthi, S. (2003). SXSLT: A manipulation language for XML. In *Proceedings of PADL*.

[Moro et al. 2002] Moro, M. M., Edelweiss, N., Zaupa, A. P., and dos Santos, C. S. (2002). TVQL - Temporal Versioned Query Language. In *International Conference on Database and Expert Systems Applications, DEXA, 13.*, volume 2453 of *Lecture Notes in Computer Science*, pages 618–627, Aix-en-Provence, France. Berlin: Springer-Verlag.

[Moro et al. 2001a] Moro, M. M., Saggiorato, S. M., Edelweiss, N., and dos Santos, C. S. (2001a). Adding time to an object-oriented versions model. In *International Conference on Database and Expert Systems Applications, DEXA, 12.*, volume 2113 of *Lecture Notes in Computer Science*, pages 805–814, Munich, Germany. Berlin: Springer-Verlag.

[Moro et al. 2001b] Moro, M. M., Saggiorato, S. M., Edelweiss, N., and dos Santos, C. S. (2001b). Dynamic systems specification using versions and time. In *International Database Engineering & Applications Symposium, IDEAS*, pages 99–107, Grenoble, France. Los Alamitos: IEEE Computer Society.

[Noy and Musen 2004] Noy, N. F. and Musen, M. A. (2004). Ontology versioning in an ontology management framework. *IEEE Intelligent Systems*, 19(4):6–13.

[OCaml 2006] OCaml (2006). Objective caml. `http://www.caml.org`.

[Robbes and Lanza 2005] Robbes, R. and Lanza, M. (2005). Versioning systems for evolution research. In *Proceedings of IWPSE*, pages 155–164.

[Siméon and Wadler 2003] Siméon, J. and Wadler, P. (2003). The essence of XML. In *POPL*, pages 1–13.

[Thiemann 2002] Thiemann, P. (2002). WASH/CGI: Server side web scripting with sessions and typed compositional forms. In *Proceedings of PADL*.

[Vagena et al. 2004] Vagena, Z., Moro, M. M., and Tsotras, V. J. (2004). Supporting branched versions on xml documents. In *Proceedings of RIDE*, pages 137–144.

[W3C XML Work Group 2006] W3C XML Work Group (2006). XML - extensible markup language. `http://www.w3.org/XML`.

[Welsh et al. 2002] Welsh, N., Solsona, F., and Glover, I. (2002). SchemeUnit and SchemeQL: Two litle languages. In *Proceedings of Workshop on Scheme and functional programming*.

[Westfechtel et al. 2001] Westfechtel, B., Munch, B. P., and Conradi, R. (2001). A layered architecture for uniform version management. *IEEE Trans. Software Eng.*, 27(12):1111–1133.

[Wrembel and Morzy 2005] Wrembel, R. and Morzy, T. (2005). Multiversion data warehouses: Challenges and solutions. In *Proceedings of ICCC*, pages 137–144.

[Wright and Felleisen 1994] Wright, A. K. and Felleisen, M. (1994). A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94.

[Zamulin 2003] Zamulin, A. V. (2003). Formal semantics of the ODMG 3.0 object query language. In *ADBIS*, pages 293–307.