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 \textit{Versioned Object 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 recover 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 \textit{Versioned Object Data Model} (VODM), a class-based object model based on ODMG \cite{Cattell2000} 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 \textit{attributes}, \textit{relationships}, and \textit{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 \textit{versioned object} in charge of grouping all versions of the same object. Each versioned object has also a \textit{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 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_teaches_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 antecedors/successors of a version).
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
current version. Additionally, in the notation $q \rightarrow a$, $q$ is a set of versioned objects or versions. The result of $q \rightarrow a$ 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 ($\text{is leaf}, \text{is root}$) and binary operators that compare the relative position of two versions in the version derivation tree of a versioned object ($\text{is succ}, \text{is pred}$). 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 $\text{Person}$ has only one instance. Class $\text{Faculty}$ has one versioned object with two versions. Finally, class $\text{Course}$ 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 $\text{FacultyM}$ represents the extension of the class $\text{Faculty}$. 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...
in this query, \( f \) ranges over the current version in the extent FacultyM. Adding the word \texttt{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 \texttt{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 \texttt{CS114} and \texttt{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 → 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:

\[ vee \in VEE = ExtId \xrightarrow{\text{Ext}} P(Vo_id) \]

**Versioned object environment.** The state of a versioned object (vobjstate) is given by a tuple \((c, \text{versions}, \text{current}, \text{deriv})\) 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*.

\[ \text{vobjstate} \in \text{Vobjstate} = \text{ClassId} \times \mathcal{P}(\text{Vid}) \times \text{Vid} \times \mathcal{P}(\text{Vid} \times \text{Vid}) \]

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 \(\text{vo_id}\) to a pair \((c, \text{vobjstate})\) where \(c\) and \(\text{vobjstate}\) are the versioned object’s class name and state respectively. We use \(\text{voe}\) as a metavariable for versioned object environments.

\[ \text{voe} \in \text{VOE} = \text{Vo_id} \leftarrow \text{ClassId} \times \text{Vobjstate} \]

**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:

\[ k \in \text{Stored} \]

\[ k ::= b \quad \text{booleans} \]

\[ | \quad i \quad \text{integers} \]

\[ | \quad s \quad \text{strings} \]

\[ | \quad \text{vo_id} \quad \text{versioned object ids} \]

\[ | \quad \text{v_id} \quad \text{versions ids} \]

\[ | \quad \{k_1, \ldots, k_n\} \quad \text{sets} \quad n \geq 0 \]

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

\[ \text{vstate} \in \text{VersionState} = \text{AttrId} \rightarrow \text{Stored} \]

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

\[ \text{ve} \in \text{VE} = \text{V_id} \leftarrow \text{ClassId} \times \text{VersionState} \]

Moreover, a database state is represented by a triple \((\text{vee}, \text{voe}, \text{ve})\) 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:

\[
v \in Values \\
v ::= b \quad \text{booleans} \\
| i \quad \text{integers} \\
| s \quad \text{strings} \\
| v_{id} \quad \text{version ids} \\
| \{v_1, \ldots, v_n\} \quad \text{set of values} \quad n \geq 0
\]

In order to reduce the number of rules we use evaluation contexts. An evaluation context \(E\) 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:

\[
E ::= \bullet \\
| \{v, E, q\} \\
| E \, bop \, q \\
| v \, bop \, E \\
| uop \, E \\
| E.a \\
| E->a \\
| if \, E \, then \, q_1 \, else \, q_2 \\
| select \, q_a \, from \, E \, x_1, \ldots, q_n \, x_n \, where \, q_b \\
| select \, q \, from \, where \, E \\
| E->versions
\]

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:

\[
vee; voe; ve \vdash q \rightarrow q' \\
vee; voe; ve \vdash E[q] \rightarrow E[q'] \\
\]

(\text{Ctx})
4.2 Evaluation Rules

We now 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`.

\[
\text{vee; voe; ve} \vdash \text{select } q_a \text{ from } \{ \} \ x_1, \ldots, q_m \ x_n \text{ where } q_b \rightarrow \{ \} (\text{Select1})
\]

The cartesian product is given by `Select2` that uses the substitution operation 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.

\[
\text{vee; voe; ve} \vdash \text{select } q_a \text{ from } \{ v_1, \ldots, v_m \} \ x_1, \ldots, q_n \ x_n \text{ where } q_b \rightarrow
\]
\[
(\text{select } q_a \text{ from } q_2 \ x_2, \ldots, q_n \ x_n \text{ where } q_b)[x_1 := v_1]
\]
\[
\text{union}
\]
\[
\ldots
\]
\[
\text{union}
\]
\[
(\text{select } q_a \text{ from } q_2 \ x_2, \ldots, q_n \ x_n \text{ where } q_b)[x_1 := v_m]
\]  

(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`.

\[
\text{vee; voe; ve} \vdash \text{select } q_a \text{ from } \text{ where false} \rightarrow \{ \} (\text{Select3})
\]

\[
\text{vee; voe; ve} \vdash \text{select } q_a \text{ from } \text{ where true} \rightarrow \{ q_a \} (\text{Select4})
\]

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

\[
\text{vec}(x) = \{ \text{vo_id}_1, \ldots, \text{vo_id}_2 \}
\]

\[
\text{vee; voe; ve} \vdash x \rightarrow \{ \text{vo_id}_1, \ldots, \text{vo_id}_2 \} (\text{VAR})
\]

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

---

\textsuperscript{3} The notation \(q[x := v]\) denotes the query that results from the substitution of \(v\) for all free occurrences of \(x\) in \(q\)
Operators. A version derivation tree of a versioned object is a set of pairs of version identifiers. If a pair \((v_{id1}, v_{id2})\) belongs to this set, it means that the version \(v_{id1}\) is an immediate predecessor of version \(v_{id2}\). Next, the premises of rules \(\text{Pred1}\) and \(\text{Pred2}\) check whether \(v_{id1}\) is predecessor of \(v_{id2}\) in the version derivation tree.

\[
\text{voe}(v_{id}) = (c, \text{versions}, \text{current}, \text{tree})
\]

\[
\text{vee}; \text{voe}; \text{ve} \vdash v_{id} \rightarrow \text{current}
\]  

(\(\text{VO}_{\text{ID}}\))

\[
\text{voe}(v_{id}) = (c, \text{versions}, \text{current}, \text{tree})
\]

\[
\text{vee}; \text{voe}; \text{ve} \vdash v_{id1} \text{ is pred } v_{id2} \rightarrow \text{true}
\]  

(\(\text{Pred1}\))

\[
\text{voe}(v_{id}) = (c, \text{versions}, \text{current}, \text{tree})
\]

\[
\text{vee}; \text{voe}; \text{ve} \vdash v_{id1} \text{ is pred } v_{id2} \rightarrow \text{false}
\]  

(\(\text{Pred2}\))

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 \(\text{Dot1}\) accesses the content of an object attribute.

\[
\text{vee}(v_{id}) = (c, \text{vstate})
\]

\[
\text{vee}; \text{ve} \vdash v_{id}.a \rightarrow k
\]  

(Dot1)

Rules \(\text{Arrow1}\) and \(\text{Arrow2}\) evaluate queries of the form \(\{v_{id1}, \ldots v_{id_n}\} \rightarrow a\) \((n \geq 0)\).

\[
\text{vee}; \text{voe}; \text{ve} \vdash \{v_{id1}, v\} \rightarrow a \rightarrow \{v_{id1}.a\} \cup \{v\} \rightarrow a \quad (\text{Arrow1})
\]

\[
\text{vee}; \text{voe}; \text{ve} \vdash \{\} \rightarrow a \rightarrow \{\} \quad (\text{Arrow2})
\]

Finally, rule \(\text{Versions}\) collects all versions of a versioned object by accessing the versioned object state.

\[
(c_i, \text{versions}_i, \text{current}_i, \text{tree}_i) \in \text{Image}(\text{voe})
\]

\[
\text{vee}; \text{voe}; \text{ve} \vdash v_{id_i} \in \text{versions}_i
\]  

\[
\text{vee}; \text{voe}; \text{ve} \vdash \{v_{id_i}\} \rightarrow \text{versions} \rightarrow \bigcup_{i=1}^{n} \text{versions}_i
\]  

(\(\text{Versions}\))
5 Type System

Types in VOQL are given by the following grammar:

\[ \sigma \in Type \]
\[ \sigma ::= \text{int} \mid \text{bool} \mid \text{string} \mid c \mid \text{set}(\sigma) \]

A type judgement for a VOQL query \( q \) has the form

\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve} ; \Gamma \vdash q : \sigma \]

where \( \text{vee}, \text{voe} \) and \( \text{ve} \) are extent, versioned object and version environments respectively, \( \text{vsch} \) 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 \( \text{vsch};\text{vee};\text{voe};\text{ve};\Gamma \) 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):

\[ \Gamma(x) = \sigma \]
\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash x : \sigma \quad \text{(VAR)} \]

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 \( x_1 \ldots x_n \).

\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash q_1 : \text{set}(\sigma_1) \]
\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma, x_1 : \sigma_1 \vdash q_2 : \text{set}(\sigma_2) \]
\[ \vdots \]
\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma, x_1 : \sigma_1, \ldots, x_{n-1} : \sigma_{n-1} \vdash q_n : \text{set}(\sigma_n) \]
\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma, x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash q_a : \text{bool} \]
\[ x_1 \text{ to } x_n \text{ different } x_1, \ldots, x_n \not\in \text{Dom}(\Gamma) \quad n \geq 0 \]
\[ \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash \text{select } q_a \text{ from } q_1, \ldots, q_n \text{ where } q_b : \text{set}(\sigma_a) \quad \text{(T-SELECT)} \]

### Versioned objects

Rules **T-VOID** and **T-VID** access the versioned object environment and the version environment to obtain the class name.
\[
\begin{align*}
\text{voe}(\text{void}) &= (c, \text{vobjstate}) \\
K \vdash \text{void} : c & \quad (\text{T-VOID}) \\
\text{ve}(\text{vid}) &= (c, \text{vstate}) \\
K \vdash \text{vid} : c & \quad (\text{T-VID})
\end{align*}
\]

Operators. Rule T-PRED simply requires that both operands of \texttt{is\_pred} be of the same class type \(c\). The rules for other positional operators are similar:

\[
K \vdash q_1 : c \quad K \vdash q_2 : c \\
K \vdash q_1 \texttt{is\_pred} q_2 : \text{bool} & \quad (\text{T-PRED})
\]

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 \(\sigma\).

\[
\begin{align*}
K \vdash q : c & \quad \text{vsch}(c) = (s, e, t) \\
\text{ty}(a) &= \sigma \\
K \vdash q.a : \sigma & \quad (\text{T-DOT})
\end{align*}
\]

Observe that the type of a query \(q \rightarrow a\), given by rule T-ARROW, is \(\text{set}(\sigma)\), where \(a\) is of type \(\sigma\) according to the schema.

\[
\begin{align*}
K \vdash q : \text{set}(c) & \quad \text{vsch}(c) = (s, e, t) \\
\text{ty}(a) &= \sigma \\
K \vdash q \rightarrow a : \text{set}(\sigma) & \quad (\text{T-ARROW})
\end{align*}
\]

The rule T-VERSIONS ensures that the construction \texttt{versions} will be applied only to collections of versioned objects.

\[
K \vdash q : \text{set}(c) \\
K \vdash q \rightarrow \text{versions} : \text{set}(c) & \quad (\text{T-VERSIONS})
\]

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 \texttt{extends}:

\[
\begin{align*}
& c \texttt{ extends } c' & (\text{EXTENDS}) \\
& c <: c' \\
& \sigma_1 <: \sigma_2 \\
& \text{set}(\sigma_1) <: \text{set}(\sigma_2) & (\text{<:SET})
\end{align*}
\]

Rule T-SUB connects the subsumption relation \(\leq\) with the type system.

\[
K \vdash q : \sigma_1 \\
\sigma_1 <: \sigma_2 \\
K \vdash q : \sigma_2 & \quad (\text{T-SUB})
\]
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

\[ \text{vsch} \vdash \text{vee, voe, ve} \]

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

\[- \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash q : \sigma, \text{ and} \]

\[- \text{vsch} \vdash \text{vee, voe, ve}, \]

then either

\[- q \in \text{Values}, \text{ or} \]

\[- \text{there is } q' \text{ such that } \text{vee; voe; ve} \vdash q \rightarrow q'. \]

**Theorem 2 (Type Preservation)** If

\[- \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash q : \sigma, \]

\[- \text{vee}; \text{voe}; \text{ve} \vdash q \rightarrow q', \text{ and} \]

\[- \text{vsch} \vdash \text{vee, voe, ve}, \]

then \( \text{vsch}; \text{vee}; \text{voe}; \text{ve}; \Gamma \vdash q' : \sigma. \)

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.](image)

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](http://www.inf.ufrgs.br/~rma/ctvql).
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,

\[\text{Figure 6: Screenshot of VOQL implementation.}\]
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. [Simeon 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


