Towards a Unified Visual Database Access

K. Vadaparty Y.A. Aslandogan G. Ozsoyoglu
Department of Computer Engineering and Science,
Case Western Reserve University, Cleveland, OH 44106
{kumarv,aslan,tekin}@alpha.ces.cwru.edu

Abstract

Since the development of QBE, over fifty visual query languages have been proposed to facilitate easy database access. Although these languages have introduced some very useful paradigms, a number of these have some severe limitations, such as: (a) not extending beyond the relational model (b) not considering negation and safety, formally (c) using ad hoc *constructs, with no analysis of expressivity or complexity done, etc. Note that visual database access is an important issue being revisted, with the emergence of different flavors of object-oriented databases. We believe that there is a need for developing a unified visual query language.*

Specifically, our goal is to develop a visual query language that has the following properties: (i) It has a few core *constructs using which "expert-users" can define new* (derived) *constructs easily (ii) "Normal users" can use easily either the core or the derived constructs for database querying (iii) It can implement representative constructs of other (textual or visual) query language straightforwardly, and (iv) It has formal semantics, with its theoretical properties, such as complexity, analyzed.*

We believe that we make a first step towards the above goal by introducing a new logical construct called restricted universal quantifier and combining it with the hierarchical structure of windows to develop a Visual Query Language, *called* **VQL**. *The core constructs of* **VQL** *can encode easily a number of* representative *constructs of different (about six visual and four non-visual) relational, nested and object-oriented query languages. We also study the theoretical aspects such as safety, complexity, etc., of* **VQL**.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0357...$1.50

1 Introduction

Since the development of QBE, over fifty visual languages (forty three surveyed in [BCCL91], and twelve in [OH93]) have been proposed. Although these visual languages enjoy some desirable properties, they have severe limitations: except a few (just 4 of [BCCL91]) the rest do not extend to object-oriented models, most do not consider formal issues involved in negation and safety, many use *ad hoc* commands/reserved words with no formal basis, etc. Moreover, with the development of object-oriented models, this issue of visual querying is being re-visited. Diverse visual languages may emerge using object-oriented query languages such as [CKW89, LO91, KKS92, BCD90, INV91] as basis. Thus, we believe that there is a need towards unifying the efforts of visual query language development. *The main contribution of this paper is the development of a visual query language*, called **VQL** (Visual Query Language), that we believe takes a first step towards this unification, because of its following properties:

- extends gracefully from the relational and nested models to disparate object-oriented models
- simulates in a user-friendly manner, the *ad hoc* commands/reserved-words of diverse object-oriented (visual and non-visual) query languages
- has formal semantics with proper consideration for negation, quantification, safety, etc.
- has provably minimal set of language-primitives while simulating desired features such as nesting, MIN, MAX, etc., and has its formal properties such as data complexity [CH82, Var82] analyzed.

VQL achieves the above by combining the power of a formal rule-based language (formal-component) with the power of nested windows (visual-component).

The relational fragment of the formal component of **VQL** is obtained by extending Datalog with a single, but crucial construct called *restricted universal quantifier*. This is extended to the object-oriented models by adding merely variables for denoting sets/complex-objects, and object-ids. *Despite the few constructs that it has,* **VQL** *is very versatile; it simulates representative features of different languages*: RC/S [OW89], STBE [OMO89], XSQL [KKS92], O_2 [BCD90], ORION [KKD89], G+ [Cru89], DOODLE [Cru92] and Hilog [CKW89] (see Section 5). Interestingly, while these languages use specialized operations such as "ALL", "GROUP-BY", "SOME", "OID FUNCTION OF", etc., to encode the representative queries we consider, **VQL** uses the same few core-constructs throughout.

The above is possible because diverse query-features such as set-comparisons [OW89, OMO89] schema querying [CKW89], path-expressions [KKS92], filtering [BCD90], *grouping* of LDL [BNST91], restricted quantification of [Kup90], *negation*, MAX, MIN, etc, are natural consequences of the core-constructs of **VQL**. Recall that grouping plays a crucial role in set based models, for different languages achieve this differently: [BNST91] introduces an explicit construct, [AG88, CKW89, LO91] use data functions or their variations, O_2[BCD90] and XSQL[KKS92] use ad-hoc constructs, STBE[OMO89] uses window-hierarchy, etc. **VQL**, on the contrary, does not require any special constructs for grouping, as the *restricted universal quantifier* simulates grouping easily. This makes us believe that **VQL** is in the right direction towards obtaining a synthesis in visual querying. We would like to emphasize that our restricted universal quantifier is more powerful than that of [Kup90] as the latter can not encode [Kup90] *negation* or *grouping*.

This paper is organized as follows. Section 2 introduces the relational fragment of **VQL** and the *restricted universal quantifier*. Section 3 shows how **VQL** extends to nested data models by encoding the *grouping* construct using the restricted universal quantifier. Section 4 gives a formal definition of the relational fragment of **VQL**, and shows how the restricted universal quantification can easily encode negation. Section 5 shows how **VQL** extends for object-oriented models; since there are only a few visual languages for object-oriented query languages, we *also* take complicated examples from the non-visual declarative languages for querying object-oriented databases and show how these can be encoded easily in **VQL**. Section 6 discusses the related work, and compares a number of visual/non-visual languages with **VQL**.

2 VQL – Relational Model

The relational fragment of **VQL** consists of Datalog extended with the *restricted universal quantifier*. This simple extension lends itself into a powerful language as the following example illustrates. In the sequel, "_" represents an unused attribute of a predicate.

Example 2.1 (Wanted Items)
Consider the query "*Get those items that are sold by every department in the second floor*" in the schema:

$DEPT(dname, manager, dvname, floor)$
$SELL(dname, iname)$
$ITEM(iname, color, price, itype)$
$SUPPLY(sname, iname, dname)$

Let $Df2$ denote the set of departments in the second floor. Then, the expression (I) given below

$\{i \mid Item(i, \_, \_, \_)\ and\ (\forall d \in Df2)\ [Sell(d, i)]\ \}$ (I)

represents the desired query.
The visual representation (Figure 1) of this query in **VQL** has two windows: *Output* and $Df2$.

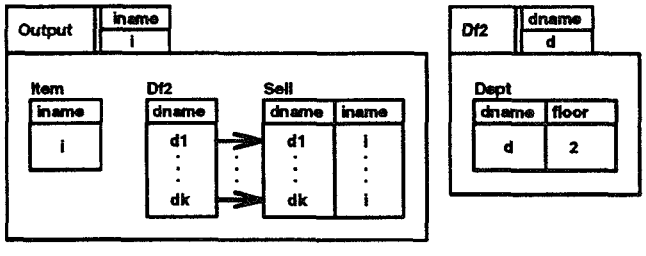

Figure 1: The Wanted Items.

Each window in **VQL** consists of a header and a body. The header consists of the window name, a list of attribute names, and their corresponding values; these can be viewed as the interface (input/output) to the window, analogous to the head of a Datalog rule. In the body of a window there may be skeletons for base or derived relations [1]. Within each skeleton there can be either an example element or a constant (represented as a string such as 2 above). In Figure 1, *Item* and *Sell* are base relations, and $Df2$ is a derived relation. The window $Df2$ computes the departments

[1] Unused attributes of a relation are not shown in the window-skeletons.

that are in the second floor. In other words, it implements the following rule:

$$Df2(x) \longleftarrow DEPT(x, -, -, 2) \qquad (II)$$

The window named *Output* has two components: the first one is a relation skeleton indicating the simple fact that i is an *Item*. The second one (which has two skeletons connected by arrows) specifies

"for every department $d_1, \ldots, d_k$ which appears in $Df2$, it is the case that $Sell(d_j, i)$ holds".

This component is the visual representation of the "restricted universal quantifier", whose formal syntax is given later. This encodes the expression

$$\text{“ } \forall d\ [Df2(d) \rightarrow Sell(d, i)]\text{”} \qquad (III)$$

Thus, the two windows $Df2$ and *Output* together represent the desired query (because (III) and (II) encode the expression (I) above). The windows are represented by the following rules in the formal component of **VQL**:

$$Output(i) \longleftarrow Item(i),\ \frac{\forall dept}{Df2(dept)}\ (SELL(dept, i))$$

$$Df2(x) \longleftarrow DEPT(x, -, -, 2)$$

In the above $\frac{\forall dept}{Df2(dept)}\ (SELL(dept, i))$ constitutes the formal syntax of the *restricted universal quantifier*, which captures the formula (III) above.

Note that if we are to use Datalog for computing the previous query, we will have to use the following rules with double (nested) negation.

$$BAD(i) \longleftarrow Item(i), Dept(d, \_, \_, 2), \neg SELL(d, i)$$

$$Output(i) \longleftarrow Item(i), \neg BAD(i)$$

Since the query is non-monotonic (in the sense of [Ull90]), positive Datalog can not encode this query while the restricted universal quantifier encodes it in a natural way. Representing, in **VQL**, recursion and rules with the same head but different bodies is discussed in [VAO92].

3 VQL – Nested Model

The relational fragment of **VQL** extends straightforwardly to nested models by a mere addition of variables that refer to nested values. Since nested values are collections, variables denoting nested values label skeletons just as predicate-names do. The most crucial operation in a nested-model is *grouping* or *nesting*. In fact, as observed in [AK90], different nested model proposals [BNST91, AG88, CKW89, LO91, OMO89, BCD90, KKS92] differ from one another on precisely how they do grouping. We show that the restricted universal quantifier can encode *grouping* directly and easily.

Example 3.1 (Simulating Grouping)
Let $T(t, c)$ denote the fact that the teacher t teaches the course c. The following rule (1) groups the tuples of T by the teacher name, into $GroupByTeacher(t, C)$ where C is a set of all courses taught by t.

$$GroupByTeacher(t, C) \longleftarrow \frac{\forall y}{T(t,y)}\ (C(y)),\ \ \frac{\forall u}{C(u)}\ (T(t, u)) \qquad (1)$$

In the above C is a set variable, referring to a set of courses. The first part of the body in the above rule, $\frac{\forall y}{T(t,y)}\ (C(y))$, tests if every course taught by the teacher t is in C. The second part tests if C does not have any "spurious" courses. i.e., every course in C is taught by t. Clearly, together, they make C to be the group of all courses taught by the teacher t.

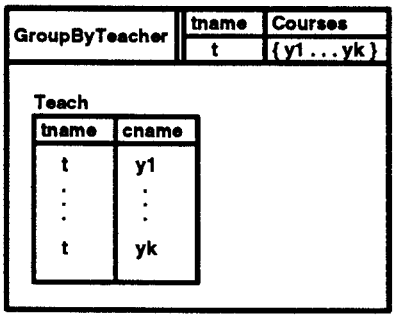

Figure 2: Grouping by teacher name.

Because grouping is an important operation, it is represented explicitly as in Figure 2 rather than as having two components corresponding to the two parts of the body in rule (1) above.

VQL, using the restricted universal quantifier can also encode MAX, MIN,etc. Let $P(p, F\_Ages)$ denote the fact that p is a parent, and F_Ages is the set of all ages of p's family. Then, the following computes $Output(p, max)$ where max is the maximum age of the members of p's family.

$$Output(pname, max) \longleftarrow P(pname, F), MAX(F, max)$$

$$MAX(F, max) \longleftarrow F(max), \frac{\forall age}{F(age)}(max \geq age)$$

4 Formal Properties

We first give syntax and semantics of the relational fragment of **VQL**, and then its formal properties: (i) restricted universal quantifier can simulate negation (ii) stratified Datalog and the stratified relational fragment of **VQL** are equal (iii) study of data complexity [Var82, CH82] of a fragment of **VQL** (this fragment involves sets). For full proofs, see [VAO92].

Definition 4.1 (Relational Fragment of VQL) *A program in* **VQL** *is a collection of rules. A rule is of the form "head* $\longleftarrow$ *body" where "head" is a positive literal, and "body" is a conjunction of literals. A literal is either (i) a positive literal, (ii) a comparison literal or (iii) a restricted-universal literal (ru-literal).*

A positive literal is of the form $P(v_1, \ldots, v_k)$ *where* P *is a* k*-ary predicate. A comparison literal is of the form* $x\theta y$ *where* $\theta \in \{=, \neq, >, <, \geq, \leq\}$*, and* x *and* y *are variables or constants. Finally, an ru-literal is a formula of the form* $\frac{\forall u}{P(u,\vec{x})}(l(\vec{y}))$ *where* P *is a predicate and* $l(\vec{y})$ *is a positive or comparison literal and* $\vec{x}$ *and* $\vec{y}$ *are vectors of variables and constants.*

The following rule computes the complement of a (monadic) relation *Availitems*; note that *Item*, the entire domain (see [Ull90]) of items, is introduced for safety.

$$NegAvailitems(x) \longleftarrow Item(x), \frac{\forall y}{Availitems(y)}(y \neq x)$$

See Figure 3 for a visual formulation of the above query in **VQL**. Many of the languages described in

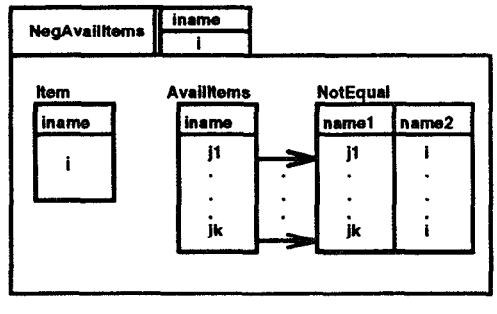

Figure 3: Simulating Negation: Unavailable items.

[BCCL91] do not have a formal notion of negation, and few discuss the issue of safety [Ull90]. First note that in **VQL** the universal quantifier is range restricted, and hence the usual (safety) problems of the universal quantifier [Ull90] do not arise. However, some queries can still be unsafe, due to the builtin implication in the ru-literal. For example, in the rule

$$Output(x) \longleftarrow \frac{\forall y}{P(x,y)}(R(y))$$

the body of the rule holds vacuously if P is empty; this in turn makes the variable x is then unbounded, and produces infinite output. This can be avoided by defining rules to be limited as follows:

Definition 4.2 (Limited Rules) *A rule is limited if every variable appearing in the head is limited in the body. A variable appearing in a positive literal in the body is limited in the body. If* x *is limited in the body and* $x = y$ *appear in the body, then* y *is limited in the body.*

Using the above definition, we show [VAO92], that a limited rule produces safe (i.e., finite) output. Since **VQL** has recursion, and can simulate negation, the natural question is how to restrict the interplay between recursion and negation. We first define a notion of "occurs positively (negatively)" and then define a notion of stratification.

Definition 4.3 (Stratification ([Llo84])) *A predicate* P *occurs positively in the formula* $P(\vec{t})$ *and in* $\frac{\forall \vec{u}}{P(\vec{u})}(R(\vec{u}))$ *where* R *is some predicate different from "*$\neq$*" and "*P*". Next,* P *occurs negatively in the formula* $\frac{\forall \vec{u}}{R(\vec{u})}(P(\vec{u}))$ *and in* $\frac{\forall x}{P(x,y)}(x \neq u)$.

A set of rules is stratified if we can assign to each predicate a positive integer (called level) so that for any rule, the level of the head predicate is greater or equal to the level of any predicate occurring positively in the body; furthermore, the level of the head predicate is strictly more than the level of any predicate occurring negatively in the body.

Using the above notion of stratification, the following can be shown (for full proofs, see [VAO92]).

Theorem 4.1 *Stratified Datalog and* **VQL** *are expressive complete for each other.*

The minimal model semantics of relational fragment of **VQL** follows easily now. The minimal model associated with a given program is what we call the output of a program on the input database. Suppose D is a database and R is a single rule in **VQL**. Then $T_R \uparrow D$ is the result of applying the rule R on the facts of D. Here the operator T_R is similar to the "T operator" defined in [Llo84]. The following theorem ensures that the output of a stratified **VQL** program Ψ can be obtained just as in the case of stratified Datalog.

Theorem 4.2 (Operational Semantics) *Given any stratified* **VQL** *program Ψ and a database D, the output of Ψ on D can be obtained by applying (polynomial number of times) the operator T_Ψ on D.*

Next we consider the formal properties of the language when there are set variables. The formal semantics (i.e., Herbrand universe, entailment, etc.) in this context are well known. Since we have grouping (defined through restricted universal quantifier), we need to follow the approach similar to that of [BNST91] or [AG88] to obtain a notion of unique minimal model. We follow the approach of [BNST91].

An important question in this context is the data complexity [Var82, CH82] of the nested version of the language. It turns out that we can analyze this using an approach similar to that of [Vad]. By a $k-$level language we mean that fragment of **VQL** which involves generation of k-level nested sets (or tuples, or a combination). The following theorem describes the data complexity of **VQL**. In the following by exponential hierarchy we mean a sequence of complexity classes $PTIME, EXPTIME, \ldots$, where each class is obtained by an exponentiation of the previous class.

Theorem 4.3 *The data complexity of k-level* **VQL** *is complete for k'th level of the exponential hierarchy.*

5 Object Oriented Models

This section shows how **VQL** extends to object-oriented databases. It turns out that we do not need any special features to navigate object-oriented databases. We choose representative queries from different object-oriented languages[2] and show how **VQL** can represent easily all these queries; note that these queries are implemented using specialized features in the respective proposals while **VQL** uses the same few constructs it has. Thus, one can use **VQL** as a visual interface to any of the logical and/or text-based (SQL-like) object-oriented language discussed here. Hence, **VQL** can justifiably be claimed to be making a first step towards synthesizing a *unified* visual query language for object-oriented databases (of different flavors). We introduce an interesting notion of *persistent queries* (see Observation 5.1 below) which seems to capture the notion of "methods" naturally.

[2]Since only a couple of visual interfaces are designed for object-oriented databases, we *also* consider a number of SQL-type and logic-based object-oriented query languages.

In the sequel, we represent a fact such as "e is a member of the class *Employee*" by e : *Employee*. Thus, the attributes of an employee id e can be represented in our formalism by a skeleton with e : *Employee* as a header, and the attributes listed explicitly as usual.

Example 5.1 (O-O Navigation [KKS92]) Consider the schema:

Employee (Qualifications : {string},
Salary : Integer,
FamilyMembers : {Person});
Employee IsA Person;
Person(Name : string, Age : Integer,
Residence: Address,
OwnedVehicles : {Vehicle})

Consider the query: "*Find all employees with a family member who is over 20 years old.*" The corresponding XSQL query is:
SELECT X FROM Employee X
WHERE $(X.FamilyMembers.Age)some > 20$
The corresponding **VQL** query is in Figure 4.

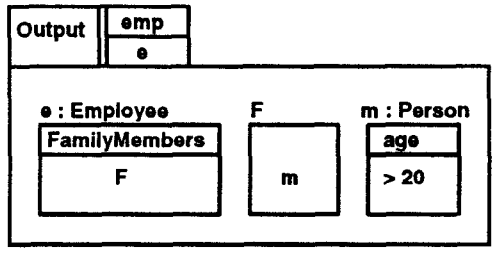

Figure 4: Employees with a family member over 20.

Figure 4 has a single window which has three skeletons: the first one specifies that e is an employee with F as the value of $FamilyMembers$. The second skeleton indicates that m is a member of F, while the third skeleton specifies m is a person with age over 20. The following is a more complex example from the same language, XSQL[KKS92].

Example 5.2 (Quantification in XSQL) Consider the query: "*Find all employees that make less than $35,000 and have family of more than four members all of which live in the same house*". The corresponding XSQL query is:
SELECT X FROM Employee X
WHERE $count(X.FamilyMembers) > 4$ *and*
$X.Salary < 35000$ *and*
X.Residence = *all* X.FamilyMembers.Residence

Note the use of "all". This construct has a special meaning: it quantifies over all ground "paths" [KKS92]; the meaning of this construct is different from the meaning of "all" used in O_2. Thus, "all" is used by different languages in different ways. **VQL** encodes this query easily as shown in Figure 5.

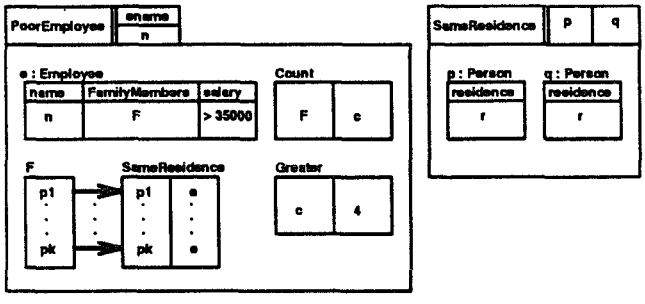

Figure 5: Poor Employees with crowded family.

Figure 5 has two windows. The window labeled *SameResidence* takes two *ids* of the class Person and determines if their residences are the same. The window *PoorEmployee* has four skeletons: the one labeled "e : *Employee*" specifies that e is an employee with salary less than 35K, *FamilyMembers* denoted by F, and *name* denoted by n. The component-skeleton (representing the restricted universal quantifier) specifies that every member $p_1, \ldots, p_k$ of e has the same residence as e. Finally, the two windows *Count* and *Greater* make sure that the cardinality of F is greater than four. The class *Employee* is a subclass of *Person*, and hence inherits the attributes *FamilyMembers* and *name*; **VQL** does this inferring for the user automatically.

Observation 5.1 (Persistent Queries)

Note that the window *SameResidence* above can be stored ("persistent query") and used over and over again, when it is needed, just like a method that determines if two employees have the same residence. We acknowledge that there is no complete agreement on the notion of "method"; for example, some authors define methods as procedures on the members of a *single* class while others [KKS92] define methods as functions with signatures that involve *more than one* class. However, a generalized approach that any "persistent query" can be viewed as a "method" seems to unify the two notions. A query can be made persistent by a user by indicating so.

In the following example taken from O_2, we show how to simulate the "filter" operation of O_2 and also to construct complex sets.

Example 5.3 (Parameterized Queries (O_2)) Consider the query (see schema in [BCD90]): "*What are the places whose activities are all open on Sundays? Give their name and their activities name and fee.* The following is the O_2 formulation of the query:

define names_and_fees (x)
select tuple(name:y.name, fee:y.fee)
from y in x;
select tuple(name: x.name,
things_to_do:
names_and_fees(x.things_to_to))
from x in Place_to_go **where for all** y **in** x.things_to_do: $y.closing\_day \neq$ "*Sunday*".

Notice that the O_2 formulation uses filters, and uses one query as an input to the other to compute the grouping. **VQL** encodes this query easily, as shown in Figure 6 with three windows: *OpenSunday*, *GetNamesFees* and *Activities*.

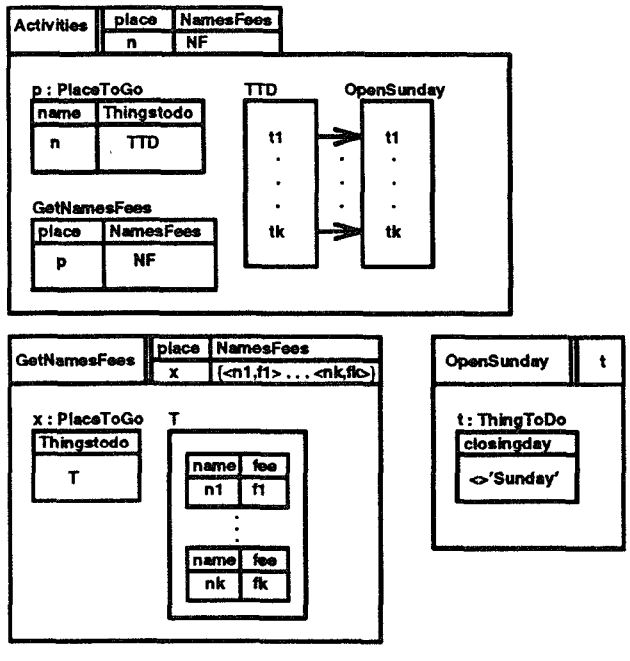

Figure 6: Sunday Activities.

OpenSunday takes a member t of *ThingsToDo* and tests if that t is open on Sundays. *GetNamesFees* takes a place x of *PlaceToGo*, and groups the names and fees of all the members of its *ThingsToDo* attribute. Note that a member of *ThingsToDo* attribute can have more than two attributes, but this query simply projects only the required two attributes

for the grouping. Finally, the window *Activities* computes the desired query, by testing if every member of *ThingsToDo* is open on sunday (for this it uses the restricted universal quantifier).

In the next example we consider a complex query from the graphical language of ORION [KKD89]. This shows how we can encode cyclic queries in **VQL**, illustrating its versatility.

Example 5.4 (ORION – Blue Cars)
The representation of the query[KKD89] "*Give all the blue cars driven by the president of the company that manufactures them*" is shown in Figure 7.

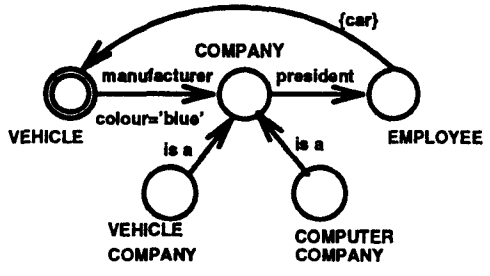

Figure 7: Blue Cars, the **ORION**'s Version.

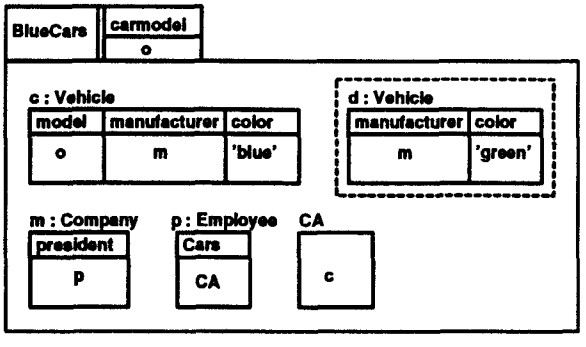

Figure 8: Blue Cars, **VQL** version (*Ignore* the dotted rectangle).

Figure 8, on the other hand, shows a representation[3] of the same query in **VQL**: first skeleton encodes that c is a vehicle with color as 'blue', model as o, and 'm' as manufacturer. The second window encodes that p is the president of m; the third window encodes that the same president, p, as an employee, owns a set of vehicles CA, and the fourth states that CA includes c. It was shown[Cru89] that a variation of the above example can not be represented in the graphical language of ORION. This is because of the multiple occurrences of a node (*Vehicle*). We give both the solution proposed by Cruz in G^+ and **VQL** solution.

[3]Ignore the skeleton inside the dotted rectangle which will be useful for the next example.

Example 5.5 (A Variation of "Blue-Cars" [Cru89])

Consider the following variation of the above example proposed by Cruz[Cru89]. "*Get all the blue cars that are driven by the president of the company that also manufactures green cars.*" Figure 9 shows the solution proposed by Cruz using G^+. The solution of **VQL** for this query involves a simple and natural change to that of the previous one – namely, add one more skeleton of *Vehicle* whose color is 'green' and the manufacturer is 'm'. Figure 8 with the skeleton in the dotted-rectangle included, shows the **VQL**'s solution. We would like to remark that if more instances of either *COMPANY* or *VEHICLE* added, even the figure of $G+$ becomes cluttered with arcs, making it hard to understand.

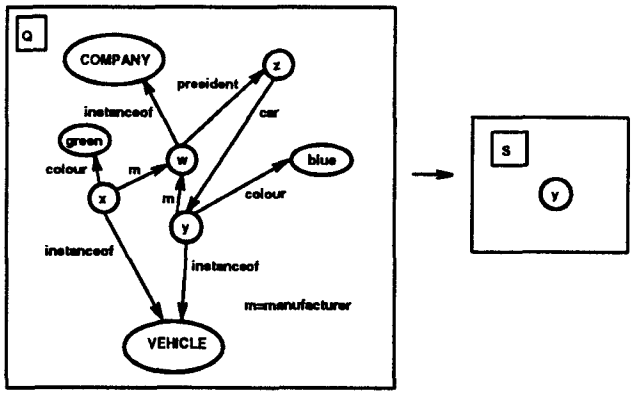

Figure 9: A variation of "blue-cars" in $G+$

5.1 Schema Querying

A very useful feature, missing in most visual query languages, is the ability to query schema (also called *meta querying*) information such as classes, attributes, etc. Krishnamurthy and Naqvi [KN88] introduced a *decidable* fragment of second order logic for meta querying. This idea was further extended by Hilog [CKW89] and XSQL[KKS92]. However, the approaches taken by [KKS92, CKW89] yield queries that are not well-typed. Below we show how we can use **VQL** to perform schema querying, both well-typed and not well-typed.

Example 5.6 (Schema Querying [KKD89]) Consider the query: "*Obtain all the class names that involve an attribute called WonNobelPrize*". We show two ways to encode the above query in **VQL**: one is identical to that of XSQL (or, Hilog) and the other involving a class called *DbClass*. Figure 10.(a) shows the untyped encoding analogous to XSQL (and Hilog); this is not well typed because the range of the variable x is not bounded. Figure 10.(b) shows the typed version with a builtin predicate *DbCLASS*. Here, DbClass is a built-in class which has names of all the classes in the database.

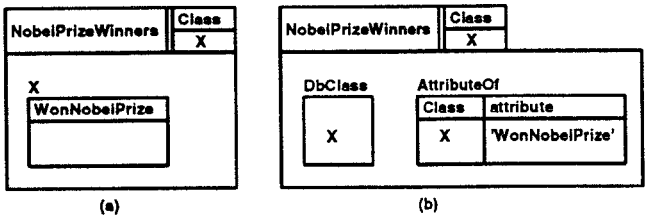

Figure 10: Schema Querying: (a) Untyped (b) Typed.

6 Related Work & Comparison

We would like to briefly survey the visual languages proposed so far, and compare them with **VQL**. The problems of QBE are well known[OH93]: its *ad hoc* additions for negation and limited grouping have no formal semantics and are ambiguous [OH93]. New visual and/or textual languages [OW89, OMO89] abandoned the universal quantifier for the reasons of safety; instead some set operators and hierarchical structure of the query were used for meeting the needs of grouping; STBE provides a very good visual language for nested-models and has formal semantics. But, it uses set variables even in the relational model (which has only constants, and tuples), is not intended for object-oriented models (actually, predates the object-oriented models), has no recursion, and finally has high complexity for checking safety of its formulas. The language G^+ provides a very good graphical interface for databases with binary predicates. However, it allows only a special class of recursion (arbitrary recursion is not available), not clear about the semantics of negation, does not extend to nested and object-oriented models, and is unintuitive for predicates predicates with arity 3 or more. Although [KKD89] provides a graphic language for object oriented models, it is not clear what is the range of queries it can support, does not have formal semantics, important notions of negation, grouping, un-nesting, etc., can not be represented. O_2 offers no powerful visual language, and the declarative (SQL-type) language it offers uses a number of *ad hoc* reserved words such as "ALL", "GROUP-BY", etc. DOODLE[Cru92] provides a graphic user interface but it is tied to F-Logic, and thus can not be looked upon as a generic visual interface; furthermore, the notions of negation, universal quantification, etc., are not available in DOODLE. G-WHIZ [HR] (Grids with Hierarchies, Imitating Zloof) is a visual interface for the functional model [Shi81]. It is similar to QBE, with constants as the only example elements; it extends QBE for nested attributes, does not represent explicit joins (functional models do not require either); it provides grouping, but does not enable negation, arbitrary recursion, ability to navigate object-oriented databases.

Some of the above languages are represented along with some of their significant features in the table of Figure 11. In this table, the first seven languages are visual (including **VQL**), while the rest are SQL-type. As can be seen from the table, a number object-oriented features are not available (indicated by empty cells) in the visual languages, and even those available are *made available explicitly* by *ad hoc* operators/reserved words/etc., and we indicate this by a check in the table. However, the non-visual languages, on the other hand, provide most of the desired object-oriented features (such as nesting, object-navigation, parameterized querying) , directly (check) or simulated ("S"), but lack visual interfaces. We believe, that **VQL** combines both: it is visual and provides (check or "S") all of the desired object-oriented features. Furthermore, **VQL** does this with the few constructs that it has (indicated by checks).

On the theoretical front also, we believe that **VQL** achieves a synthesis: *restricted universal quantifier* simulates both grouping and negation. In the past, different proposals were made to capture universal quantification, negation and grouping.

Some rule-based languages augmented Datalog with grouping directly [BNST91, AG88] enabling set-generation or, indirectly [CKW89, LO91] producing *ids* with set-values; this was done to extend Datalog to meet the needs of nested models. On the other hand, some proposals added universal quantifiers back again to capture the needed richness of the language: Kuper [Kup90] studied the formal properties of augmenting a particular form of restricted universal quantification

Language	Exp Disj	Group By	Rest. Univ.	Min Max	Count	ALL SOME	Set Ops	Rec-ursion	Form. Sem.	Neg-ation	Param Query	Nested Output	Schema Query
STBE	✓	S	S	✓	✓	S	✓		✓	S		✓	
PASTA			Lim	✓	✓	S	S	Lim		Lim			
G+	✓			✓	✓			Lim	✓	S		✓	
DOODLE	✓	✓		?	?			✓	✓		✓	✓	✓
OrionQL	?			?	Ys	?	?	?	?			?	
G-WHIZ		S	Lim	?	?	Lim		Lim	?	Lim		Lim	
VQL	S	S	S	S	S	S	S	✓	✓	S	S	✓	✓
XSQL	✓	S	S	✓	✓	✓	✓	?	Lim	✓		✓	✓
O2QL	Ys	Ys	✓	✓	✓	✓	✓	?	Lim	✓	✓	✓	?

First seven languages are visual and next two are SQL-type.

✓: Explicitly available, Empty Slot: Not Available, Lim: Limited form available, ?: Not clear, S: Simulatable
Rest.Univ: Restricted Universal Quantifier, Set.Ops: Set Operations, Schema Query: Schema Querying Capabilities, Param.Query: Parameterized Queries

Figure 11:

and proved that it can not group arbitrary elements. Contrast this with our restricted universal quantifier.

In the case of SQL-type languages, O_2 [BCD90] and XSQL [KKS92] augmented the word "ALL" into their language (each with different semantics). Neither IQL [AK89], the formal language underlying a part of O_2's query language, nor F-Logics [KL89], the formal language underlying a part of XSQL allow this construct "ALL", and hence this addition remains an *ad hoc* addition in both XSQL and the query language of O_2.

7 Conclusion

In this paper we introduced a declarative visual language called **VQL** and showed that this language extends gracefully for databases from relational to nested and object-oriented models. The crucial operation in this language is *restricted universal quantifier*, which enables encoding the representative features of disparate object-oriented query languages (both visual and non-visual) in a user-friendly manner, providing versatility to **VQL**. Thus, **VQL** can be used as a visual interface on the top of the existing powerful non-visual languages. We also showed some interesting theoretical results of the language. First we showed that the restricted universal quantifier can simulate negation, grouping, MIN, MAX, etc. Next, we showed the relational fragment of **VQL** matches in expressivity with stratified Datalog. Finally, we showed some data complexity results of the nested language fragment of **VQL**.

We plan to extend this language in many ways. Some of these are: (i) need to extend **VQL** to account for object-generation, and other update operations (ii) Try to develop an "object code" into which **VQL** can be translated, and the programs in this object code can efficiently executed (iii) Incorporate inheritance reasoning into **VQL**. We do have some ideas in these directions. Especially, regarding the "object-code", we believe that structural recursion such as the ones discussed in [OBB89, BBN91] may be potential candidates.

8 Acknowledgments

We would like to acknowledge the comments of the reviewers which improved the presentation. Thanks to Z. M. Ozsoyoglu for a number of discussions. The first author likes to acknowledge S. Naqvi for some clarifications on the issue of meta querying. The second and third authors would like to acknowledge the partial support of the NSF grants IRI-8811057 and IRI09008632.

References

[AG88] Abiteboul, S. and Grumbach, S. Col: A

logic-based language for complex objects. In *Proceedings of Extended Database Technology (EDBT)*, pages 271–293, 1988.

[AK89] Abiteboul, S. and Kanellakis, C. P. Object identity as a query language primitive. In *Proceedings of the ACM SIGMOD International conference on Management of Data*, pages 159–173, Portland, Oregon, June 1989.

[AK90] Abiteboul, S. and Kanellakis, P. Database theory column: Query languages for complex object databases. In *SIGACT news*, 1990.

[BBN91] Breazu-Tannen, V, Buneman, P., and Naqvi, S. Structural recursion as a query language. In *Workshop on DPL*, 1991.

[BCCL91] Batini, C., Catarci, T., Costabile, M.F., and Levialdi, S. Visual query systems. Technical report, Department of Informatica E Sistemistica, Universita Degli Studi Di Roma "La Sapienza", 1991.

[BCD90] Bancilhon, F., Cluet, S., and Delobel, C. The o_2 query language syntax and semantics. Technical report, INRIA, 1990.

[BNST91] Beeri, C., Naqvi, S., Shmueli, O., and Tsur, S. Set constructors in a logic database language. *Journal of Logic Programming*, 1991.

[CH82] Chandra, A.K. and Harel, D. Structure and complexity of relational queries. *Journal of Computer Systems and Sciences*, 25(1):99–128, 1982.

[CKW89] Chen, W., Kifer, M., and Warren, D. Hilog as a platform for database languages. In *Workshop on DBPL*, 1989.

[Cru89] Cruz, I. Declarative query languages for object-oriented databases. *F.H. Lochovsky, editor, Office and Data base Systems Research*, 1989.

[Cru92] Cruz, I. Doodle: A visual language for object-oriented databases. In *Proceedings of the ACM SIGMOD*, 1992.

[HR] Heiler, S. and Rosenthal, A. G-whiz, a visual interface for the functional model with recursion. *Proceedings of VLDB 85, Stockholm.*

[INV91] Imielinski, T., Naqvi, S., and Vadaparty, K. Querying Designing and Planning Databases. In *Proceedings of the Second International Conference on Deductive Object-Oriented Databases*, Munich, Germany, December 1991.

[KKD89] Kim, K, Kim, W., and Dale A. Cyclic query processing in object-oriented databases. In *IEEE Intl. Conference on Data Engineering*, 1989.

[KKS92] Kifer, M., Kim, W., and Sagiv, Y. Querying object-oriented databases. In *Proceedings of the ACM SIGMOD*, 1992.

[KL89] Kifer, M. and Lausen, G. F-logic: A higher-order language for reasoning about objects, inheritance, and scheme. In *Proceedings of the ACM SIGMOD*, 1989.

[KN88] Krishnamurthy, R. and Naqvi, S. Towards a real horn clause language. In *VLDB*, 1988.

[Kup90] G.M. Kuper. Logic programming with sets. *JCSS*, 41(1), 1990.

[Llo84] Lloyd, J. *Foundations of Logic Programming.* Springer Verlag, 1984.

[LO91] Lou, Y. and Ozsoyoglu, Z.M. Llo: An object oriented deductive language with methods and method inheritence. In *Proceedings of the 1991 ACM SIGMOD International conference on management of data*, 1991.

[OBB89] Ohori, A., Buneman, P., and Breazu-Tannen, V. Database programming in machiavelli - a polymorphic language with static type inference. In *Proceedings of the ACM SIGMOD conference*, 1989.

[OH93] Ozsoyoglu, G. and H. Wang. Example-based graphical database query languages. *To appear in IEEE, Computer, July*, 1993.

[OMO89] Ozsoyoglu, G, Matos, V., and Ozsoyoglu, Z.M. Query processing techniques in summary table by example database query language. *Transactions of Database Systems*, 14(4), December 1989.

[OW89] Ozsoyoglu, G. and Wang H. A relational calculus with set operators, its safety and equivalent graphical languages. *IEEE Software Engineering*, 15(9):1041–1052, September 1989.

[Shi81] Shipman, D.W. The functional data model and the data language daplex. *acm TODS*, 6(1):140–173, March 1981.

[Ull90] Ullman, J.D. *Databases and Knowledge Bases*, volume 1 of *Computer Science.* Computer Science Press, 1990.

[Vad] Vadaparty, K. On the power of rule based languages with sets. In *Proceedings of PODS 1991;* extended version to be *submitted to the Journal of Logic Programming.*

[VAO92] Vadaparty, K., Aslandogan, Y., and Ozsoyoglu, G. Towards a unified visual database access. Technical Report 92-19, Case Western Reserve University, 1992.

[Var82] M. Vardi. The complexity of relational query languages. In *Proceedings of the 14th STOC*, pages 137–146, 1982.