Towards an ODMG-Compliant Visual Object Query Language

Manoj Chavda
Department of Computer Science
University of Cape Town
Rondebosch 7700, South Africa
manoj@cs.uct.ac.za

Peter T. Wood
Department of Computer Science
University of Cape Town
Rondebosch 7700, South Africa
ptw@cs.uct.ac.za

Abstract

We describe the design, implementation and user evaluation of QUIVER, a graph-based visual query language for object databases. The design goals of QUIVER include compliance to standards, comprehensive representational power, and consistency of visual representation. Compliance to standards is achieved through QUIVER queries being translated to OQL, the standard query language proposed by the Object Data Management Group (ODMG). Comprehensive representational power is gained by QUIVER supporting a significant number of object database constructs, including objects, literals, attributes, relationships, structures, collections, operations, (aggregate) functions, and subqueries. Consistency of visual representation is pursued by assigning similar visual representations to constructs with similar functionality, as well as by minimising the use of text in QUIVER queries. The language is implemented as a visual front-end to the O_2 object database system. Results of a user evaluation suggest that users find it easier to formulate correct queries in QUIVER than in OQL.

1 Introduction

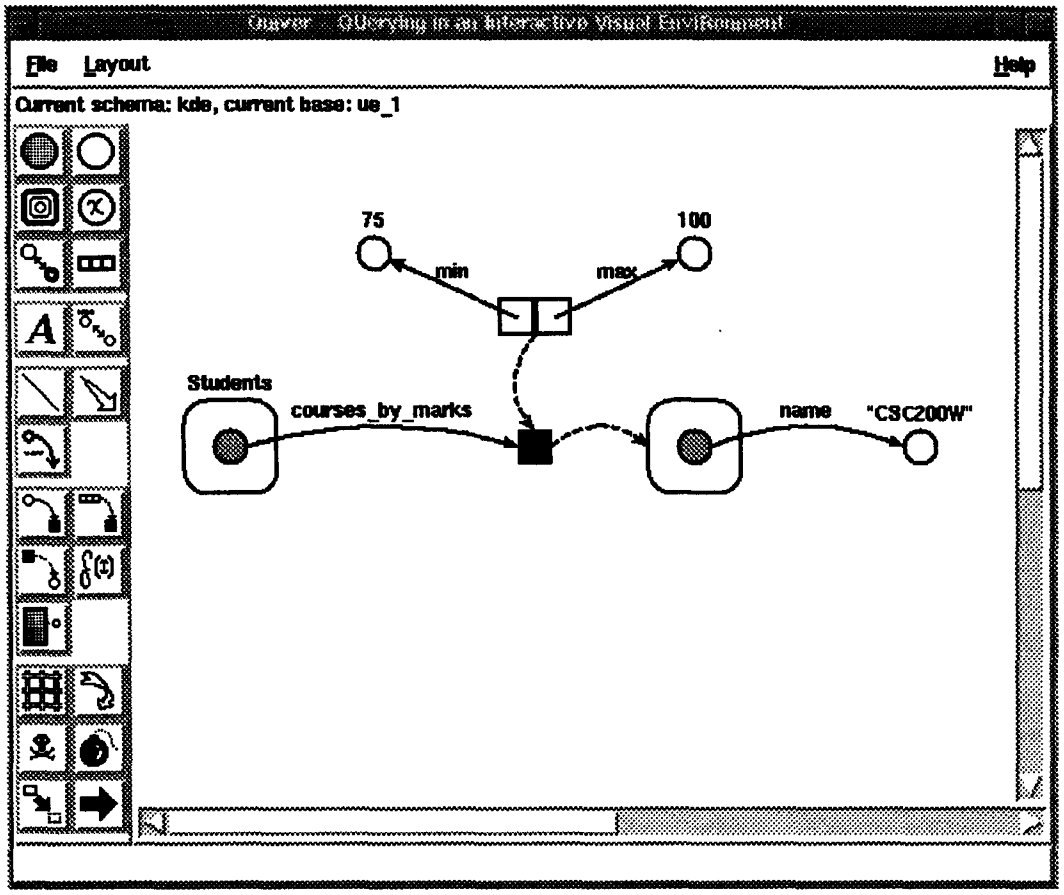
A large number of visual query languages for databases have been described in the literature over the years, each language employing a particular visual representation and being designed for a particular data model. For example, graph-based representations are used in GOOD [GPVdBVG94], Gql [PK95], GraphLog [CEH+94], Hyperlog [PL94], VDM/VDL [Orm92], and VQL [MK93], while visual query languages for object databases include OdeView [DGJS95], OOQBE [STT91], OQD [KM93], and VQL [VAÖ93]. In this paper, we describe the design, implementation and evaluation of a new graph-based visual query language for ODMG-compliant object databases called QUIVER (an acronym for "QUerying in an Interactive Visual EnviRonment").

In 1993, the Object Database Management Group (ODMG) released their first draft of a standard for object databases, called ODMG-93 [Cat96], which included the definition of an Object Query Language (OQL). This query language was essentially the same as that used in the O_2 object database [Deu90], which was also called OQL. Queries constructed in QUIVER are translated into OQL, and evaluated by the O_2 server. The answers to a query are displayed by the O_2 application O_2Look. QUIVER maintains a very loose coupling with O_2, making it easy for QUIVER to query other ODMG-compliant databases.

Graphs seem like a natural representation for object databases, a claim supported by the many forms of object diagrams used in object-oriented modeling. Indeed, QUIVER is part of a larger project to provide a consistent environment in which to visualize object database schemas, queries and instances. The desire to be able to visualize object database instances requires that we have visual representations for all ODMG object model constructs.

Figure 1 shows an example of a QUIVER query. The schema of the object database being queried defines objects such as students and courses. Each course object has an attribute called name, while an operation (method) called courses_by_marks is defined on student objects. This operation takes as input two integer parameters, min and max, and returns the collection of courses for which the student scored between min and max. The collection of all

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997



Current schema: kde, current base: ue_1

Figure 1: Did any students score between 75 and 100 in CSC200W?

student objects is called Students. The QUIVER query in Figure 1 determines whether any students scored between 75% and 100% in the course CSC200W.

In general, a QUIVER query is a nested, mixed graph, comprising nodes, (undirected) edges and (directed) arcs. The graph can be nested because a node may have other nodes (and possibly arcs) included in it. In the query Q of Figure 1 two of the nodes are *object* nodes, represented by shaded circles. Although not indicated in Q, the lefthand object node represents a student object, while the righthand object node represents a course object. Each object node is included in a rounded rectangle which represents a collection. The inclusion is represented by an arc, but the arc remains invisible. The three non-shaded circles in Q represent *literal* nodes, labeled 75, 100 and "CSC200W". The lefthand collection node represents the *extent* called Students, while the righthand one represents the result of the courses_by_marks operation. The operation itself is represented as a solid square. Since invocation of an operation requires input and produces output, there are *data-flow* arcs (represented by dashed arcs) into and out of the operation node in Q. The input to the operation is a structure node (represented by a rectangle) which includes two structure element nodes (represented by squares). The associations between the student object node and its courses_by_marks operation, between the course object and the value of its name attribute, and between the structure element nodes and their values are all represented as labeled arcs. Finally, the ordering between the two structure element nodes is represented by another invisible arc called an adjoinment arc.

Although the schema of interest to a user is not (yet) displayed graphically, the user is guided by pop-up menus during the formulation of a query on a schema. For example, clicking on the object node button (●) or the named node button (⊗) in Figure 1 will bring up a menu of all object types or all named items, respectively, defined in the current schema. A user typically starts composing a query from a named item, such as the extent named Students. To select a property of an object, the user clicks first on the *place property arc* button (↘). Now clicking on an object node will display a menu of its properties.

From Figure 1 we see that QUIVER supports objects, literals, structures, collections (sets, bags, arrays and lists), attributes and operations. In addition, QUIVER supports relationships, aggregate functions and subqueries, as well as various constraints such as equality, inequality and subset. Hence, while languages such as GraphLog [CEH+94], GOOD [GPVdBVG94], Hyperlog [PL94] and VQL [VAÖ93] advocate a minimal set of visual constructs, a distinguishing feature of QUIVER is the richness of its visual constructs.

Despite this richness, QUIVER does not currently support all OQL constructs. For example, there is no visual counterpart to the `group by` or `order by` constructs of OQL. In addition, negation and universal quantification are not supported, although universal quantifiers are used in the translation of subset constraints which are incorporated in QUIVER.

Another distinguishing feature of QUIVER is its minimal use of text. The use of text is limited to names defined in the schema being queried. In particular, unlike in a number of proposed languages [PL94, KM93, MK93], there is no need to use variables in QUIVER queries, nor is output identified textually.

In the next section we cover background material and related work. We first provide an overview of ODMG-93 in Section 2.1, before surveying a number of related visual query languages in Section 2.2. In Section 3, we present the design of the QUIVER query language, guided by a taxonomy of visual query constructs. The implementation of QUIVER is described in Section 4. We focus, in particular, on the translation of QUIVER queries to OQL. In Section 5, we present a user evaluation comparing QUIVER queries and OQL queries in terms of the speed of query construction and the ease with which correct queries can be formulated. The results of the evaluation allow us to conclude that users find it easier to construct correct queries in QUIVER than in OQL. Finally, concluding remarks and directions for future work are discussed in Section 6.

2 Background

2.1 ODMG-93

In this subsection we give very brief overviews of the object model proposed by the Object Database Management Group (ODMG) and its object query language (OQL).

The basic modeling primitive in the ODMG object model is the *object*. Objects have state and behaviour, and can be categorized into *types*. All objects of a certain type have common behaviour and a common range of states. In our university database example, there are objects of type Person, Student, Course and Department. The *extent* of a type is the set of all instances of that type. In our example, the name of each extent is simply given by the plural of the name of its associated type; hence Students is the extent of objects of type Student.

The state and behaviour of an object are collectively referred to as its *characteristics*. The state of an object is defined by the values of a set of *properties*, which are either *attributes* of the object (with literal values), or *relationships* between the object and one or more others (with object values). In our example, each object of type Person or Course has an attribute name, whose value is a string. A Student object has a relationship takes which is a set of Course objects, while a Course has a relationship is_taken_by (which has as value a set of Student objects).

The behaviour of an object is defined by a set of *operations*. In our example, Student objects have an operation courses_by_marks, which takes as input two integer parameters, min and max, and returns a set of Course objects.

Objects may be organized into a graph of subtypes and supertypes. A subtype inherits all of the characteristics of its supertypes, and may define additional characteristics. For example, Student is a subtype of Person, and so inherits the name attribute.

Individual objects or collections of objects can be given names meaningful to end-users. For example, the name `Vice-Chancellor` might be assigned to the appropriate person object in a university database.

The ODMG object query language (OQL) provides declarative access to objects. The version we use here is that which is compatible with O_2 OQL [O2T94]. Below we give two simple examples of OQL queries. The first example shows an OQL query equivalent to that in Figure 1:

```
exists x in Students:
exists y in x.courses_by_marks(75,100):
(y.name = "CSC200W")
```
(1)

The second example shows an OQL query which involves structured output, a nested subquery, and the use of an aggregate function:

```
select struct
   (student:  x.name,
    courses: (select y.name
              from y in x.takes
              where count(x.takes) > 4))
from x in Students
```
(2)

In this query, the output is a bag of tuples, each with two named components. The first component is a string representing a student name, while the second is the bag of course names taken by the student, as long as the student takes more than four courses.

Further examples of OQL queries will be presented in Section 4.1 which describes the translation of QUIVER queries to OQL.

2.2 Related Visual Query Languages

In this section, we review visual query languages which are either graph-based [GPVdBVG94, PK95, CEH+94, PL94, Orm92, MK93], or designed for object databases [DGJS95, STT91, KM93, VAÖ93]. Of these, two have been named VQL, one by Mohan and Kashyap [MK93], and the other by Vadaparty *et al.* [VAÖ93]. We will refer to them as MK-VQL and VAO-VQL, respectively.

Of the above languages, only MK-VQL [MK93] is both graph-based as well as implemented on an object database. While OQD is also graph-based, no mention is made of its implementation in [KM93]. Although GOOD [GPVdBVG94] and Hyperlog [PL94] are both defined in terms of objects, neither is in fact implemented on an object database system. GOOD is implemented on a relational database system, while Hyperlog is implemented on a functional database system. Gql [PK95] and VDM/VQL [Orm92] are designed for the functional data model, although they could probably be applied to object databases. GraphLog [CEH+94] is implemented on a deductive database system. OdeView, OOQBE and VAO-VQL [VAÖ93] are all form- or table-based rather than graph-based languages. Since MK-VQL [MK93] is implemented on an in-house object database system, QUIVER is the only ODMG-compliant graph-based query language, as far as we are aware.

Languages such as GraphLog, GOOD, Hyperlog and VAO-VQL advocate a minimal set of visual constructs for achieving substantial expressive power. In contrast, QUIVER permits a large number of visual constructs. For example, neither structures nor collections are modeled explicitly in GraphLog, GOOD or Hyperlog, while operations are not provided for by GraphLog, Hyperlog or VAO-VQL. Although Gql provides a fairly rich set of constructs which bear a number of similarities with those of QUIVER, it too does not support structures, collections or operations.

The ability to nest nodes inside other constructs is a feature QUIVER shares with GraphLog and Hyperlog. In addition, the use of boldface to indicate output in QUIVER has been borrowed from GraphLog. However, the use of data-flow arcs to allow a consistent representation of the computations performed by operations, aggregate functions and (sub)queries is unique to QUIVER.

As stated in the Introduction, there is no need to use variables in QUIVER, nor is output specified textually. In contrast for example, variables are needed in Hyperlog in order to equate items, while output is specified textually in OQD. Variables are used both for certain equality tests and for output in MK-VQL.

Recursion is provided in GraphLog, Hyperlog, GOOD and MK-VQL, while negation is provided in all the languages mentioned above. Neither recursion nor negation have yet been implemented as part of QUIVER, although we see no difficulty in doing so. Our ideas on how incorporate these features into QUIVER are given in Section 6.

3 The QUIVER Query Language

In this section, we describe the syntax of a QUIVER query; the semantics are defined in terms of a translation to OQL which is presented in Section 4.1. We begin with an informal discussion of the syntax, aided by another example of a QUIVER query.

A QUIVER query is a graph comprising nodes, arcs and edges. Each node, arc and edge has an associated visual

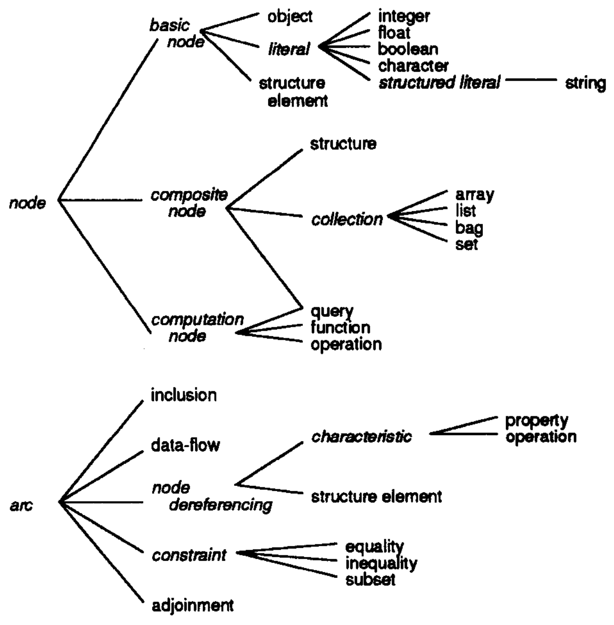


Figure 2: Hierarchy of visual types in QUIVER.

representation and denotes a QUIVER query construct. For example, nodes represented visually as shaded circles denote objects. The hierarchy of all QUIVER visual query constructs is shown in Figure 2, where instantiable types are shown upright and abstract types are shown *italicised*. For simplicity, we have classified all arcs and edges as arcs in the diagram.

As mentioned, one of the goals of the QUIVER language is that it should be consistent; similar query concepts should be represented by similar visual constructs. The types in Figure 2 are grouped by their functional similarity, each type within a group having a similar visual representation. For example, all computation nodes are represented as rectangles, all node dereferencing arcs are represented as solid, labeled arrows, and all composite nodes include other nodes.

Figure 1 introduced examples of nodes representing objects, literals, structure elements, structures, collections and operations, as well as inclusion arcs, data-flow arcs, a property arc, an operation arc, structure element arcs, and an adjoinment arc. As another example, consider the query Q in Figure 3. Query Q finds students who take more than four courses, in each case returning the name of the student along with the bag of names of the courses the student takes. Three other constructs from Figure 2 are used in Q: a function node labeled count(), a (sub)query node represented by the large rectangle, and an inequality arc labeled with $>$.

Being a computation node, a function node is represented as a rectangle, although since the computation is hidden from the user, the rectangle is filled in. On the other hand, a query node is both a computation node and a composite node (see Figure 2). Hence, it is represented as a rectangle which includes the nodes and arcs specifying the computation (see Figure 3). For the outer query, this

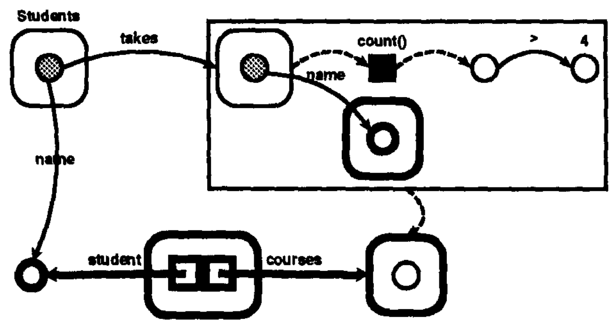


Figure 3: Finding students who take more than four courses (using a subquery).

bounding rectangle is not necessary and has been omitted in our examples.

Like an operation node, a function node has data-flow arcs entering and leaving it. Queries do not require an incoming data-flow arc as their input is implicitly the entire database. However, subqueries do require an outgoing data-flow arc to a node representing the output type of the query (see Figure 3).

Output for a query Q is specified by boldface nodes (and possibly arcs). A query containing no bold nodes, such as that in Figure 1, is interpreted simply as a constraint, and therefore produces a Boolean value as output. On the other hand, the bold subgraph of the outer query Q of Figure 3 specifies that the output of Q is a collection of structures, each having a Student object as its first component and a collection of strings as its second component.

Arcs in Q can be labeled, while edges, inclusions and adjoinments cannot. Each node in Q can have a pair of labels m and n, displayed as $m : n$, where m is the *prefix* of the node label and n is the *suffix*. In order not to clutter the query graph, *abbreviated labels* are also supported. With abbreviated labels, only the prefix of each label is displayed, as shown in the examples in Figures 1 and 3. The prefix of the label indicates the name of the object, literal or collection, if a name has been defined in the schema. Otherwise, the prefix of the label is empty for an object or collection node, while for a literal node it is either empty or the value of the literal. The suffix of the label always indicates the type of the construct represented by the node.

3.1 Syntactic Restrictions

A QUIVER query is a nested, mixed graph $Q = (N, A, E, I, J)$, where N is a set of nodes, A is a set of arcs, E is a set of edges, I is a set of inclusions, and J is a set of adjoinments. An arc a is an ordered pair (u, v) of nodes, while an edge e is an unordered pair $\{u, v\}$ of nodes. An inclusion is analogous to an arc in that it is an ordered pair (u, v) of nodes. However, the subgraph of Q induced by I, denoted Q_I, is restricted to being a forest. An adjoinment is also an ordered pair (u, v) of nodes. The subgraph of Q induced by J, denoted Q_J, is restricted to being a set of disjoint paths.

Let $Q = (N, A, E, I, J)$ be a QUIVER query. We say that node v is *included in* node u if either $(u, v) \in I$ or there is a node w such that $(u, w) \in I$ and v is included in w. A node u is *maximal* if it is not included in any node. In other words, a maximal node is the root of some tree in Q_I. Each tree in Q_I is called an *inclusion group*.

Because $Q = (N, A, E, I, J)$ is a nested, mixed graph, we need to extend the usual definitions of connectedness and connected components. If $\{u, v\} \in E, (u, v) \in A, (v, u) \in A, (u, v) \in I, (v, u) \in I, (u, v) \in J$, or $(v, u) \in J$, we say that u and v are *connected*. If u and v are connected and v and w are connected, then u and w are also *connected*. The *connected* relation is an equivalence relation which induces a partition of Q into disjoint *connected components*.

If, after deleting any output collection node used to collect answers, a query Q has more than one connected component, then Q is interpreted as a disjunctive query. All components must either be Boolean queries or must have output subgraphs (defined below) whose root nodes are of the same type.

For simplicity, let us assume from now on that query Q comprises only a single component. We have seen that queries can be nested inside one another, each query node being the root of a query inclusion group. The *outermost subquery* of Q, denoted Q_0, is the subgraph of Q induced by those nodes which are not included in any query node. For example, the outermost subquery Q_0 in Figure 3 comprises all nodes (and their incident arcs) except those included in the subquery rectangle. Note that Q_0 excludes the arc labeled takes.

Given a query node q in Q, the *subquery* associated with q, denoted Q_q, comprises the subgraph induced by the nodes v of Q such that (q, v) is an inclusion in Q, along with any arc (u, v) or edge $\{u, v\}$ where (q, u) is not an inclusion in Q. For example, the subquery associated with the query node in Figure 3 comprises all the nodes and arcs included in the query rectangle, as well as the arc labeled takes (and the student object node at its tail). Thus the student object node is in both the outermost subquery and the inner subquery.

When we refer to the *output subgraph* O of a query Q, we are referring to the subgraph induced by the bold nodes and arcs (including inclusions and adjoinments between bold nodes) of Q_0. Nested subqueries in Q may or may not have output subgraphs of their own, but these are not part of O. If O is empty, then Q is called a *Boolean query* (for example Figure 1).

There are a number of restrictions on the syntax of an output subgraph O. For our purposes, O must be acyclic, with exactly one node of indegree zero, which we call the *root* of O. We call the nodes with outdegree zero in O the *leaves* of O.

The final syntactic restriction on a subquery Q involves the notion of the *reachability* of nodes in Q. All nodes in

Q, except possibly the root of O, must be reachable. There are three sources (or roots) of reachability. A node u is *reachable* in Q, and a *reachability root* for Q, if u is

1. a named object, literal or collection node, or
2. a literal node with a value label, or
3. a reachable node which is at a level less than Q, or
4. a reachable query node.

A query node is reachable if all the non-bold nodes in Q_q are reachable. Reachability can also be passed from the tail to the head of an arc. Space does not permit us to enumerate the rules for reachability here; they can be found in [Cha96]. Instead, as an example of establishing reachability, consider the query Q of Figure 1. The collection node labeled Students in Q is reachable by virtue of (1) above, while the literal nodes labeled 75, 100 and CSC200W are reachable by virtue of (2). Since Students is reachable, so is the object node included in Students. Now we can establish that the input structure node is reachable, followed by the operation node, and the collection node c representing the output of the operation. Finally, we determine that the object node included in c is reachable.

Let Q be a subquery graph with output subgraph O. The *pattern subgraph* P of Q is the subgraph induced by the non-bold arcs, edges, inclusions and adjoinments in Q. For Q to be syntactically correct, P and O must have as intersection the leaves of O. In Figure 3, P and O have the nodes which are the targets of the bold arcs labeled student and courses in common. If there are any nodes in Q that are neither reachable nor bold, then Q is not syntactically correct. The order in which nodes in Q are determined to be reachable also provides an order for translating Q into OQL, as described in the next section.

4 Implementation of QUIVER

Figure 4 presents a diagrammatic view of the various components which make up the QUIVER application, where the arrows between components represent the flow of data. The shaded boxes represent external components which were not part of the QUIVER development. For example, O_2 (from O_2 Technology [Deu90]) is used as the persistent store of QUIVER, O_2Look is used as the data presenter, and DOT [KN93] is used for graph layout. In this paper, we present only the query translation algorithm in Section 4.1.

QUIVER is implemented using `Tcl/Tk` [Ous94] and C++. `Tcl/Tk` is used for the the graphical user interface, while C++ is used for the query translation as well as for calling and managing the interfaces between the various components. In all, QUIVER comprises approximately 15,000 lines of `Tcl/Tk` code and about 10,000 lines of C++ code.

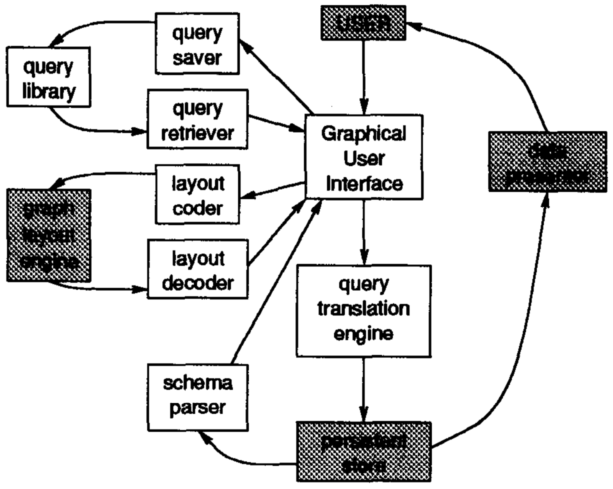


Figure 4: The components comprising QUIVER.

4.1 Query Translation

In this section, we define the semantics of QUIVER queries in terms of a translation to OQL. Through necessity, many details are omitted, a more complete description being given in [Cha96]. Here we simply present the pseudo-code for the algorithms used for query translation, Algorithms 1 and 2, and illustrate their use with a single example.

For simplicity, we assume that QUIVER query Q comprises only a single connected component. Given a QUIVER query Q, the translation of Q to an OQL expression $\mu(Q)$, obtained by calling *translateQuery* (Algorithm 1) with the outermost subquery Q_0 of Q, consists of four phases. Firstly, the `select` clause of $\mu(Q)$ is generated from the output subgraph O. Secondly, the set of variable bindings in the `from` clause of $\mu(Q)$ is generated from F, the subgraph of the reachability forest R needed to show the reachability of each leaf node in O. Thirdly, the subgraph W induced by arcs in R which are not in F is used to generate further variable bindings in the `where` clause of $\mu(Q)$. Finally, the subgraph C induced by arcs in the pattern subgraph P which are not in R is used to generate conditions to be tested in $\mu(Q)$. Of course, the output subgraph O may be empty, in which case only the third and final phases generate output, but without the keyword `where`. Similarly, either W or C may be empty. The four subgraphs O, F, W and C form a partition of the arcs of Q.

To generate the fragments of the OQL expression corresponding to F, W and C, we call a procedure *translateQueryFragment*, shown as Algorithm 2. In fact, this procedure has to generate slightly different strings depending on whether it is called with F, W or C, but we do not discuss these details here. In addition, we have not included pseudo-code for the procedures *translateOutput* called in Algorithm 1 or *translateArc* called in Algorithm 2. We have rather used an example to illustrate their operation.

Consider the query Q of Figure 3 which is translated as

translateQuery(Q)
Input: QUIVER query Q.
Output: OQL string S.
Method:

```
1. $S := T :=$ empty string
2. Mark all nodes and arcs in $Q$ as unprocessed.
3. Assign variable names to each node in $Q$.
4. Let $O$ be the output subgraph and $P$ be the pattern subgraph of $Q$.
5. Let $R$ be the breadth-first spanning forest of $P$ based on reachability.
6. Let $F$ be the subforest of $R$ establishing reachability of leaves of $O$.
7. Let $W$ be the subforest of $R$ induced by arcs not in $F$.
8. Let $C$ be the subgraph of $P$ induced by arcs not in $R$.
9. if $O$ is not empty
10.    then
11.      begin
12.        if root of $O$ is reachable
13.          then append "element(select" to $S$ and ")" to $T$
14.          else
15.            begin
16.              append "select" to $S$
17.              if root of $O$ is a set then append "distinct" to $S$
18.            end
19.        translateOutput($O$)
20.        append "from" to $S$
21.        translateQueryFragment($F$)
22.        if $W \cup C$ not empty then append "where" to $S$
23.      end
24. if $W$ is not empty
25.    then translateQueryFragment($W$)
26. if $C$ is not empty
27.    then translateQueryFragment($C$)
28.    else append "true" to $S$
29. append $T$ to $S$
30. if some node or arc in $Q$ is not processed then fail
```

Algorithm 1: Pseudo-code for query translation.

translateQueryFragment(Q)

```
1. for each unprocessed node $u$ in $Q$ with name or value $v$ do
2.   begin
3.     append "$var(u)$ in bag($v$)" to $S$
4.     mark $u$ processed
5.   end
6. while the set of processed nodes changes do
7.   begin
8.     for each unprocessed arc $(u, v)$ in $Q$ such that $u$ is processed do
9.       begin
10.        translateArc($(u, v)$)
11.        mark $(u, v)$ and $v$ processed
12.      end
13.    for each unprocessed, translatable query node $q$ in $Q$ do
14.      begin
15.        append "$var(q)$ in bag(" to $S$
16.        translateQuery($Q_q$)
17.        append ")" to $S$
18.        mark $q$ processed
19.      end
20.  end
```

Algorithm 2: Pseudo-code for translating a query fragment.
follows:

```
select struct(
       student: VAR2_1, courses: VAR4_2)
from VAR1_2 in bag(Students),
     VAR1_1 in VAR1_2,
     VAR2_1 in bag(VAR1_1.name),
     VAR3_1 in bag(
       select VAR6_1
       from VAR5_2 in bag(VAR1_1.takes),
            VAR5_1 in VAR5_2,
            VAR6_1 in bag(VAR5_1.name)
       where
         exists VAR8_1 in bag(count(VAR3_2)):
        (exists VAR9_1 in bag(4):
        (VAR8_1 > VAR9_1))),
     VAR4_2 in bag(VAR3_1)
```
(3)

The translation of Q begins by assigning a variable name, derived from a numbering of the inclusion groups in Q, to each node in Q. For example, the node at the head of the arc labeled student is assigned the variable VAR2_1, while that at the head of the arc labeled courses is assigned the variable VAR4_2. The `select` part of expression (3) is generated from the output subgraph O of Q.

Since all of the nodes in the pattern subgraph of the outermost subquery in Q are needed to establish the reachability of the output nodes, there is no `where` clause in the outermost OQL expression in (3) above. For the `from` clause, assume that the collection node c_1 labeled Students is assigned the variable VAR1_2, while the object node o_1 included in c_1 is assigned the variable VAR1_1. Since c_1 is named, line 3 of Algorithm 2 binds VAR1_2 to Students using `VAR1_2 in bag(Students)`.

Now o_1 is reachable, so *translateArc* in Algorithm 2 binds VAR1_1 to objects included in VAR1_2, using

```
VAR1_1 in VAR1_2
```

Next the output object node (VAR2_1) becomes reachable and is bound to VAR1_1.name.

The final part of Algorithm 2 which needs to be explained is that dealing with the translation of subqueries. As can be seen on line 16 of Algorithm 2, subqueries are translated by a recursive invocation of the procedure *translateQuery*. However, a subquery can only be translated if its corresponding query node is *translatable*. We say that a query node q is *translatable* if all those nodes, which are in both Q_q and another subquery, have been processed. This ensures that the variables corresponding to these nodes have been bound in the OQL expression by the time the subquery is translated. These nodes now become roots of reachability for the subquery.

Returning to the translation of query Q of Figure 3, the inner subquery becomes translatable once object node o_1 (VAR1_1) has been processed. The subquery generates a `where` clause in expression (3), since the function node, its output node (VAR8_1) and the literal node (VAR9_1) labeled 4 in Q are not needed to establish the reachability of the output nodes of the subquery in Q.

The OQL expression (3) should be compared to the much shorter expression (2) given in Section 2.1. The essential difference is in the overhead of binding variables to named collections or literals in (3). A rewriting optimization could, of course, remove this overhead. Note also that the subquery appears in the `from` clause in expression (3), while it appears in the `select` clause in expression (2).

The user can save the OQL expression generated by QUIVER in a file by typing control-Q after formulating a query. Another possibility, which we have not explored, would be to display the OQL expression incrementally as the user formulates their query.

5 A User Evaluation of QUIVER and OQL

In order to compare the usability of QUIVER and OQL, we conducted a limited user evaluation. The evaluation was performed by thirteen fourth-year computer science students, each of whom was given a list of eight queries to construct. The queries ranged from simple retrievals to complex queries requiring both long path traversals and subqueries.

The thirteen participants were divided into two groups: seven participants in GROUP Q used QUIVER, while six participants in GROUP O used OQL. None of the participants had any prior experience with QUIVER or OQL, although four participants in GROUP Q and three in GROUP O had prior experience in using SQL from a typical undergraduate database course. Two introductory lectures (one on QUIVER and one on OQL) were presented to the participants.

Two hypotheses were tested. The first was that OQL queries can be constructed as easily as QUIVER queries, in other words, that GROUP O participants would attempt the same number of queries as GROUP Q participants. The second hypothesis was that GROUP Q participants would construct a similar number of correct queries to GROUP O participants. The statistical test used for both hypotheses was the *Mann-Whitney-Wilcoxon* procedure [Gib76]. This test was chosen because the data arises from two independent samples and, because of the small sample size, we cannot assume that the data follows a normal distribution.

For the first hypothesis, let M_Q represent the mean of the number of queries attempted in GROUP Q, and M_O represent the mean of the number of queries attempted in GROUP O. The null hypothesis H is that $M_O = M_Q$, while the alternative A is that $M_O > M_Q$. The results of our evaluation show that we can conclude A only with 71% confidence. Since this offers no convincing evidence that H is false, we conclude that QUIVER queries can be constructed just as easily as OQL queries.

For the second hypothesis, we wish to determine if participants in GROUP Q obtained more correct queries than participants in GROUP O. For this purpose, participants' queries were classsified as either correct, essentially correct (a minor mistake or two), or incorrect. For this hypothesis we used the ratio r of the number of queries classified correct or essentially correct to the total number of queries attempted. Let M_Q represent the mean ratio of GROUP Q, and M_O represent the mean ratio of GROUP O. The null hypothesis H is that $M_Q = M_O$, while the alternative A is that $M_Q > M_O$. The results of our evaluation show that we can conclude A with 99.7% confidence. That is, it is easier to construct correct QUIVER queries than it is to construct correct OQL queries.

There are a number of possible sources of error in our user evaluation. Firstly, participants were not selected voluntarily, nor did they all have identical backgrounds. Secondly, participants were taught QUIVER and OQL in only one 45-minute session each. Finally, a larger sample size would have produced more accurate results.

Nevertheless, the results of the second hypothesis are extremely encouraging, as they show that QUIVER does aid the user in producing more correct queries. We believe this is particularly the case for queries involving long paths of references and for nested queries. In both cases, the visual nature of QUIVER allows the user to see these constructs more clearly. In addition, the fact that a QUIVER user does not have to remember which variable ranges over which collection may reduce the number of errors when compared to an OQL user.

6 Conclusion

We have described the design, implementation and evaluation of QUIVER, a visual query language for ODMG-compliant object databases. QUIVER is comprehensive in the sense that it includes visual representations for a large number of the modeling constructs found in object databases. This is in contrast to many other visual query languages which have concentrated on using a minimal number of representations. Nevertheless, a serious attempt has been made to provide a consistent organisation of the QUIVER visual representations, as indicated by the taxonomy presented in Section 3.

Our user evaluation comparing QUIVER with OQL was particularly encouraging, in that it showed that it is easier to construct correct queries with QUIVER than it is with OQL. This may be partly because keeping track of paths and the ranges of variables is difficult in an OQL expression. In QUIVER, the paths are visual and there are no variables.

There is no doubt that QUIVER can be both improved and extended. The query language could be improved by introducing greater conciseness. Examples include removing the necessity for the output of subqueries to be duplicated (although this duplication is performed by the interface), and allowing paths to be specified by means of regular expressions, as in **G** [CMW87]. This would allow recursive queries to be formulated in QUIVER, although the translation could then no longer generate simply OQL. Another extension to QUIVER would be the introduction of negation, for which we are considering the use of crossed out arcs, as in GraphLog [CEH+94], for example.

The current translation of QUIVER queries could be made more efficient by not introducing variables for every node, and by recognising that certain special cases could be translated without needing to resort to a `select ...from` expression.

Finally, a more extensive user evaluation of QUIVER may uncover further areas for improvement. For example, a number of users in our limited evaluation suggested that the simultaneous display of the OQL query while a QUIVER query was being constructed would be beneficial.

Acknowledgements

We are grateful to O_2Technology for the use of O_2 during the development of QUIVER, and to the Foundation for Research Development for financial support.

References

[Cat96] R. G. G. Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. Morgan Kaufmann, San Francisco, 1996.

[CEH+94] M. P. Consens, F. Ch. Eigler, M. Z. Hasan, A. O. Mendelzon, E. G. Noik, A. G. Ryman, and D. Vista. Architecture and applications of the Hy$^+$ visualization system. *IBM Systems Journal*, 33(3):458–476, 1994.

[Cha96] M. Chavda. Visually querying object-oriented databases. Master's thesis, Department of Computer Science, University of Cape Town, Rondebosch 7701, South Africa, 1996.

[CMW87] I. F. Cruz, A. O. Mendelzon, and P. T. Wood. A graphical query language supporting recursion. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 323–330, 1987.

[Deu90] O. Deux *et al.* The story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, March 1990.

[DGJS95] S. Dar, H. Gehani, V. Jagadish, and J. Srinivasan. Queries in an object-oriented graphical interface. *Journal of Visual Languages and Computing*, 6(1):27–52, March 1995.

[Gib76] J. D. Gibbons. In I. Olkin, editor, *Nonparametric Methods for Quantitative Analysis*, chapter 4, pages 159–173. Holt, Rinehart and Winston, 1976.

[GPVdBVG94] M. Gyssens, J. Paredaens, J. Van den Bussche, and D. Van Gucht. A graph-oriented object database model. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):572–586, August 1994.

[KM93] J. C. Kwak and S. Moon. Object query diagram: An extended query graph for object-oriented databases. In *Proc. IEEE Symposium on Visual Languages*, pages 44–48, 1993.

[KN93] E. Koutsofios and S. C. North. DOT: a directed graph layout engine. AT&T Bell Laboratories, Murray Hill, NJ, October 1993.

[MK93] L. Mohan and R. L. Kashyap. A visual query language for graphical interaction with schema-intensive databases. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):843–858, October 1993.

[O2T94] O_2 Technology. *Object Query Language OQL Manual*, Version 4.5, November 1994.

[Orm92] L. Orman. A visual data model. *Data & Knowledge Engineering*, 7(3):227–238, February 1992.

[Ous94] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[PK95] A. Papantonakis and P. J. H. King. Syntax and semantics of Gql, a graphical query language. *Journal of Visual Languages and Computing*, 6(1):3–25, March 1995.

[PL94] A. Poulovassilis and M. Levene. A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems*, 12(1):35–68, January 1994.

[STT91] F. Staes, L. Tarantino, and A. Tiems. A graphical query language for object oriented databases. In *Proc. IEEE Workshop on Visual Languages*, pages 205–210, 1991.

[VAÖ93] K. Vadaparty, Y.A. Aslandogan, and G. Özsoyoglu. Towards a unified visual database access. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 357–366, May 1993.