VOQL* : 귀납적으로 정의된 형식 시맨틱을 지닌 시각 객체 질의어

(VOQL* : A Visual Object Query Language with Inductively-Defined Formal Semantics)

이 석 균 †

(Suk Kyoon Lee)

요 약 객체 데이타베이스를 위해 최근에 제안된 VOQL(Visual Object Query Language)은 경로식과 집합 관련 조건을 시각화하고 형식 시맨틱을 제공하는데 성공적이었다. 그러나 기존의 VOQL은 몇가지 문제들이 있다. VOQL에서는 집합 관련 조건만이 허용되고, 변수 개념을 위한 명시적인 언어 구문이 없어서 질의문들은 종종 어색하고 직관적이지 못하다.

본 논문에서는 이러한 문제들을 극복하도록 VOQL을 확장한 VOQL*를 제안한다. 시각변수 개념을 도입하고 이에 기초하여 VOQL의 문법과 시맨틱을 수정하였다. VOQL*의 언어 구문들은 OOPC의 문법을 반영하도록 조심스럽게 정의되어서, 시각변수, 시각요소, VOQL* 단순 텀(term), VOQL* 구조 텀, VOQL* 기본 포뮬라(formula), VOQL* 포뮬라, VOQL* 질의문 등의 VOQL* 구문들이 OOPC 구문들처럼 계층적이고 재귀적으로 구성되어 있다. 가장 중요한 것은 VOQL*의 각 구문의 시맨틱이 OOPC를 사용하여 재귀적 방법을 통한 형식 정의가 이루어진다는 점이다. 잘 정의된 문법과 시맨틱으로 말미암아, VOQL*의 질의문들은 명확하고 간결하고 직관적이다. 또한 VOQL* 질의문을 OOPC 질의문으로 번역하는 효과적인 절차를 제공한다. VOQL*는 로직의 문법적 구조를 반영하는 잘 정의된 문법과 재귀적 방법으로 형식 시맨틱을 제공하는 첫번째 시각 질의어이다.

Abstract The Visual Object Query Language (VOQL) recently proposed for object databases has been successful in visualizing path expressions and set-related conditions, and providing formal semantics. However, VOQL has several problems. Due to unrealistic assumptions, only set-related conditions can be represented in VOQL. Due to the lack of explicit language construct for the notion of variables, queries are often awkard and less intuitive.

In this paper, we propose VOQL*, which extends VOQL to remove these drawbacks. We introduce the notion of visual variables and refine the syntax and semantics of VOQL based on visual variables. We carefully design the language constructs of VOQL* to reflect the syntax of OOPC, so that the constructs such as visual variables, visual elements, VOQL* simple terms, VOQL* structured terms, VOQL* basic formulas, VOQL* formulas, and VOQL* query expressions are hierarchically and inductively constructed as those of OOPC. Most important, we formally define the semantics of each language construct of VOQL* by induction using OOPC. Because of the well-defined syntax and semantics, queries in VOQL* are clear, concise, and intuitive. We also provide an effective procedure to translate queries in VOQL* into those in OOPC. We believe that VOQL* is the first visual query language with the well-defined syntax reflecting the syntactic structure of logic and semantics formally defined by induction.

· This work was supported by Korea Science And Engineering Foundation (KOSEF) through Advanced Information Technology Research Center(AITrc).

† 종신회원 : 단국대학교 전산통계학과 교수

sklee@dankook.ac.kr

논문접수 : 1999년 11월 22일

심사완료 : 2000년 4월 10일

1. Introducion

Rapid progress in information technology on WWW, GIS, CAD/CAM, multimedia, and etc. demands the capability of managing complex structured data in databases. Object data models to meet the challenge have been proposed [2, 3, 15, 5]. In these models, data are grouped into sets of objects called classes according to their properties and behaviors. Classes are organized into the class aggregation and class generalization hierarchies.

The complexity of data structures in the underlying data model requires appropriate expressive capability of associated query languages be powerful enough to manipulate data and formulate queries effectively. The complexity of data structures increases the level of complexity of associated query languages and, in turn, places a heavier cognitive load on database users. In order to reduce such load on users and to maintain the expressive capability required by the underlying model, visual query languages are considered as a solution. However, research on visual query languages for object data models is still in a primitive stage.

Many visual query languages [1, 6, 7, 8, 10, 11, 16, 18] have been proposed. QBD* [1], G+ [7], ERC [8], and GRAQULA [18] cannot represent object-oriented features such as *path expressions* and various set comparisons, since they are based on relational or entity-relationship (ER) models. The notion of the path expression [4, 12, 9], which denotes a nested structured data object spreading across classes by a sequence of attribute names and dot notations, provides a compact expressive capability of representing and managing complex data objects in the class hierarchies. A path expression may simply be viewed as a sequence of relationship instances joined together without an explict join notation.

In visual query languages for object data models[6, 11, 16, 19], queries are constructed on a sub-graph of a schema diagram. However, path expressions in VQL [16] and VQL [19] are represented by adding textual variables to a sub-graph of a schema diagram. Since the textual variables are used to link relationship instances implied by the sub-graph, they fail to graphically visualize the sequence of relationship instances linked together across several classes [13, 14]. With all formal syntax and semantics, GOOD [11] only considers data modification operations such as deletions, additions, and updates. QUIVER [6], which is capable of representing relationship instances visually, lacks formal syntax and semantics. Therefore, it fails to properly address issues, such as path expressions, related to object database query languages.

The *visual object-oriented query language* (VOQL) recently proposed in the references [13, 14] has been designed to handle set-related comparisons visually and represent path expressions in the tree-structured graph. VOQL has the formal semantics based on the Object-Oriented Predicate Logic (OOPC) [4], which is an extension of the tuple relational calculus. However, VOQL assumes that every attribute is multi-valued. Due to this assumption, all the intermediate results in path expressions are represented by the set notation in VOQL, even though they are single-valued. It makes queries less intuitive and difficult to understand.

VOQL does not explicitly include the notion of variables as a basic language construct. The absence of the explicit language construct for the notion of variables in VOQL causes the semantics of other language constructs to become complicated since other constructs implicitly take the role of variables. Moreover, various issues related to the notion of variables, such as quantifiers, scoping problems, and explicit joins, are not easy to address, and the solutions for them are often awkward and complex.

In this paper, we propose VOQL* which has significantly improved VOQL in many ways. The language constructs in VOQL* are designed carefully to reflect corresponding ones in OOPC. The notion of the *visual variable* is introduced to correspond to the *object variable* in OOPC, and the notion of the *visual element* to represent the value of a single-valued attribute. A visual element is usually an intermediate result or final result of a *VOQL* structured term* explained below. Introducing the notion of visual

variables allows us to treat universal quantification and existential quantification in the same way as in traditional logic.

Based on visual variables and visual elements, the notions of the *VOQL* simple term* and *VOQL* basic formula* are presented corresponding to the notions of *term* and *atomic formula* in OOPC. The collection of VOQL* simple terms may be organized into the tree-structured *VOQL* structured term*. A VOQL* structured term, also called the *VOQL* path expression,* represents the collection of OOPC textual path expressions with the same object variable. The *VOQL* basic formula* is a condition consisting of VOQL* simple terms and comparison operators. The *VOQL* formula* is a collection of VOQL* basic formulas, whose VOQL* simple terms constitute some VOQL* structured terms. *VOQL* query expressions* consist of VOQL* formulas and target lists, which represent the attributes to project.

The semantics of each language construct in VOQL* are formally defined inductively using OOPC. The semantics of queries in VOQL* can easily be derived through inductive application of the semantics of the language construct. Moreover, we provide an effective procedure to translate queries in VOQL* into those in OOPC.

This paper is organized as follows. In Section 2, we introduce basic language constructs of VOQL* and simple examples. We then present the syntax and semantics of VOQL* in Section 3. We illustrate more complex query expressions in Section 4. Finally, we end with closing comments and discussion on future research issues in Section 5.

2. Basic Visual Constructs and Examples in VOQL*

In this section, we begin with the description of OOPC, and then, illustrate several VOQL* query examples. We provide the semantics of VOQL* query examples using OOPC.

2.1 OOPC and Path Expressions

Among many object-oriented query languages such as XSQL [12], O_2SQL [2], and PathLog [9], we choose OOPC for its simple syntax, intuitive semantics, and formal foundation. The OOPC is an extension of tuple relational calculus, and the details can be found in the reference [4].

A typical OOPC expression has a structure similar to that of an SQL statement;

{Target list | Range clause; Qualification clause}.

The *target list* specifies what must be retrieved. The *range clause*, written as v/C where v is an object variable and C a class name, specifies the range of the object variable v, i.e., the set of objects of the class C to which v is bound. Note that an object variable appearing in the target list should be bound to some class in the range clause. The *qualification clause* is a well-formed formula. Predicates in the formula are either atomic or quantified. An atomic predicate is *true, false,* a range clause, or $t_1\ \theta\ t_2$, where θ is a comparison operator, and t_i a constant, an object variable v, or an attribute value $v.Att_j$ Note that a range clause is not included in the definition of atomic predicates in the reference [4]. However, since a range clause x/C is more often written as x∈C and is considered an atomic predicate in many database literatures, we include it here. Set-related operators such as ∈ and ⊆ are allowed in addition to comparison operators in tuple relational calculus. A quantified predicate is either ∀*Range Clause* (*Qualification clause*) or ∃*Range Clause* (*Qualification clause*).

As an example, let's consider the following query example written in OOPC.

OOPC 2.1 {x.Name | x/Company; ∃y/Employee [(y = x.President) ∧ (y.Salary > $100K)] },

where x and y are object variables bound to the classes Company and Employee in the range clauses, respectively.

The range clause and qualification clause of OOPC may be interpreted as the conjunction of the former and the latter; the range and qualification clauses as a whole are called the *OOPC well-formed formula* (*wff*). For instance, x/Company [∃y/Employee ((y = x.President) ∧ (y.Salary > $100K))] in OOPC 2.1 is

an OOPC wff.

In order to provide navigational capability over complex structures of objects reflecting class hierarchies, OOPC allows the path expression [4, 9, 12, 17]-- a term constructed from an object variable and a sequence of attribute names. The idea is to follow a sequence of links between objects without having to specify explicit join conditions. Consider the following OOPC query:

OOPC 2.2 {*x*.Model | *x*/Vehicle;
x.Manufacturer.Type = Corporation ∧
x.Manufacturer.Headquarter.City = "New York"}

x.Manufacturer.Headquarter.City and *x*.Manufacturer.Type are path expressions where *x* ranges over the class Vehicle. The dot function in the path expression plays the role of function composition (e.g., City(Headquarter(Manufacturer(*x*)))). In this paper, we follow the definition of path expressions in ODMG 2.0 [5], where a multi-valued attribute is allowed only in the last position in a path expression.

2.2 Visual Constructs and Query Examples In VOQL*

The basic language constructs in VOQL* are *blobs, subblobs, directed edges, visual variables, visual elements, stump blobs with undirected edges,* and *labels.* As in Fig. 1 and 2, blobs are represented by rectangles, subblobs by nested rectangles, visual variables by shaded circles, visual elements by transparent circles, stump blobs by shaded ecllipses. Blobs represent classes (including subclasses) with extensional notion. A blob represents the set of objects in a class or a subclass. A subblob represents a subset of a blob. Visual elements represent objects in classes. Objects in user-defined classes are represented by visual elements enclosed by blobs, while those in primitive domains by visual elements without enclosing blobs.

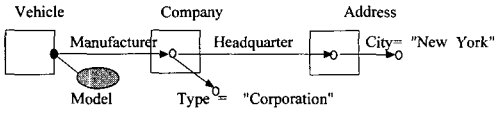


Fig. 1 The VOQL* expression for OOPC 2.2.

A visual variable is always located at the side of a blob or a subblob. A pair of a variable and a blob (or a subblob) to which the variable is attached, represents that the variable is bound to the set of objects in the blob (or the subblob). Directed edges, which are used to represent attributes, are always placed either between elements or from variables to elements. Labels attached to directed edges are the names of corresponding attributes. Conditions in VOQL* are often represented by comparisons between a textual constant and a visual element which represents an attribute value. Stump blobs denoted by shaded ellipses, connected through undirected edges to visual variables or elements represent the attributes to be projected. Let us reconsider the query in OOPC 2.2. The corresponding visual query in VOQL* is given in Fig. 1.

Note how the target list, range clause, and qualification clause of OOPC 2.2 are visualized in the VOQL* query expression in Fig. 1. Let v1, e1, e2, e3, e4, sb1 denote the visual variable in the blob Vehicle, the visual element in the blob Company, the visual element in the blob Address, the visual element labeled with "City", the visual element labeled with "Type", and the stump blob labeled with "Model", respectively. The range clause *x*/Vehicle is represented by the blob Vehicle and the variable v1. The target list *x*.Model is represented by the variable v1, the stump blob sb1 and the undirected edge between v1 and sb1.

The qualification clause in OOPC 2.2 consists of the equality comparison between the path expression "*x*.Manufacturer.Headquarter.City" and the constant value "New York". The path expression "*x*.Manufacturer.Headquarter.City" is visualized as a sequence of the variable v1, the edge labeled with "Manufacturer", the element e1, the edge labeled with "Headquarter", the element e2, the edge labeled with "City", and the element e3. The variable v1 and the elements e1, e2, e3 represent the textual path expressions *x*, *x*.Manufacturer, *x*.Manufacturer.Headquarter, and *x*.Manufacturer.Headquarter.City, respectively. Note that a visual element represents an intermediate or final result of some path expression.

Let's consider another VOQL* expression in Fig. 2 corresponding to the expression of OOPC 2.3.

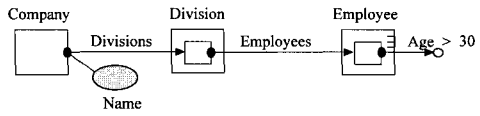


Fig. 2 The VOQL* expression for OOPC 2.3.

OOPC 2.3 {*x*.Name | *x*/Company; ∃*y*/Division [*y* ∈ *x*.Divisions ∧ ∃*z*/Employee [*z* ∈ *y*.Employees ∧ *z*.Age > 30]]}

For multi-valued attributes such as Divisions and Employees, subblobs are used instead of elements. Let v1, sb1, v2, sb2, v3, and e1 denote the visual variable for the blob Company, the subblob and the visual variable for it inside the blob Division, the subblob and the visual variable for it inside the blob Employee, and the rightmost element in Fig. 2. The target list in OOPC 2.3 is represented by the variable v1 and the stump blob labeled with "Name" ; the range clause by the variable v1 and the blob Company. The subblobs sb1 and sb2 represent sets of objects that x.Divisions and y.Employees return, respèctively. Note that v2 and v3 are bound to sb1 and sb2, which are a subset of Division and a subset of Employee, respectively. In general, a visual variable *v*, the existential quantifier near the variable *v*, and the blob (or the subblob) *b* to which the variable *v* is bound, represent ∃*x*/*set* where *v* is the visual representation of the textual variable *x*, and *b* is the visual representation of the range *set*.

3. VOQL* : Syntax And Semantics

In this section, we formally define the syntax of VOQL* based on Higraph. Similar to that of OOPC, the syntax consists of VOQL* terms, VOQL* formulas, and VOQL* expressions. We then inductively define the semantics of VOQL* using OOPC. For each language construct in VOQL* such as VOQL* terms, VOQL* formulas, and VOQL* expressions, we provide corresponding language constructs in OOPC. The semantics of queries in VOQL* are defined by translating each language construct in VOQL* into corresponding language constructs in OOPC and combining them according to the rules. Note that the translation process is also inductive from the language constructs at a lower level to the ones at a higher level of the syntactic structure.

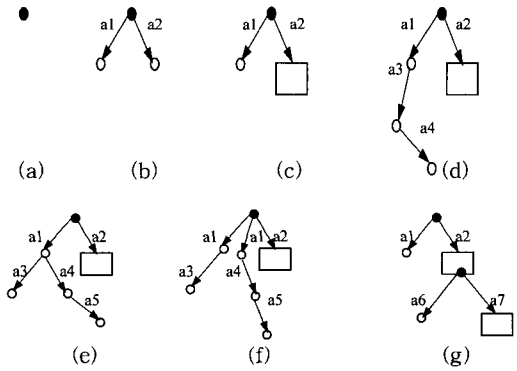


Fig. 3 VOQL* structured terms

3.1 VOQL* Terms

Blobs, subblobs, visual elements, visual variables, and textual constants including text strings and numbers, are *VOQL* simple terms*. Based on VOQL* simple terms, *VOQL* structured terms* are defined. VOQL* structured terms have a tree structure: the root of a tree is a visual variable, internal nodes are visual elements, and leaf nodes are either visual elements or subblobs. Nodes are connected by directed edges labeled with attribute names. Note that subblobs are allowed only as leaf nodes of a VOQL* structured term.

Examples of VOQL* structured terms are illustrated in Fig. 3. Trees in Fig. 3 (a)-(f) are all single VOQL* structured terms. Case (a) shows the simplest VOQL* structured term. The depth of a node in a VOQL* structured term is the number of directed edges from the root to the node. The depth of a VOQL* structured term is the largest of depths of nodes in the tree. The depths of structured terms in Fig. 3 (a)-(f) are 0, 1, 1, 3, 3, and 3, respectively.

The example in Fig. 3 (g) is not a single VOQL*

structured term. It shows two VOQL* structured terms participating in a VOQL* formula which will be explained in Section 3.2. The visual variable of the VOQL* structured term at the bottom is bound to the set of objects represented by the subblob of the VOQL* structured term at the top. The query expression in Fig. 2 has three VOQL* structured terms whose depths are all one. The path from the root of a tree T to a node N in T is represented by the string resulting from concatenating the variable name of the root and the sequence of dots and labels attached to directed edges from the root to N. The resulting string is the OOPC path expression for the node N.

Every node in a VOQL* structured term can be translated into an OOPC path expression. In translating VOQL* structured terms into OOPC path expressions, it is assumed that distinct object variables are used for each VOQL* structured term. If we visit in preorder all nodes of VOQL* structured terms in Fig. 3 (a)-(g) and convert them into OOPC path expressions, the result will be as follows:

(a) x1
(b) x2, x2.a1, x2.a2
(c) x3, x3.a1, x3.a2
(d) x4, x4.a1, x4.a2, x4.a1.a3, x4.a1.a3.a4
(e) x5, x5.a1, x5.a2, x5.a1.a3, x5.a1.a4, x5.a1.a4.a5
(f) x6, x6.a1, x6.a1, x6.a2, x6.a1.a3, x6.a1.a4, x6.a1.a4.a5
(g) x7, x7.a1, x7.a2, x8, x8.a6, x8.a7

For example, the VOQL* structured term in Fig. 3 (e) visualizes six OOPC path expressions. Introducing the tree structure into the notion of path expressions provides us the more compact and natural way of representing path expressions than OOPC does. Note that the object for each node in a VOQL* structured term is functionally determined by the object that the root (the visual variable) is bound to.

There may be VOQL* structured terms that are structurally different from, but semantically identical to each other. The VOQL* structured term in Fig. 3 (e) is structurally different from the one in Fig. 3 (f), but is semantically equivalent. The latter has redundant visual constructs -- a visual edge and a visual element.

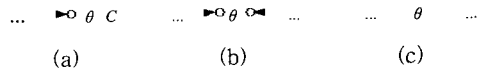
As shown above, by visualizing various relationships between textual path expressions, a VOQL* structured term has a compact representation of many textual path expressions. Since the term path expression is widely used, VOQL* structured terms are often called *VOQL* path expressions*.

3.2 VOQL* Formulas

In this section, we define VOQL* formulas. Instead of providing a BNF-like formal syntax, we describe how to construct VOQL* formulas from VOQL* terms and various comparison operators. First, we define generic forms of *VOQL* basic formulas*. We then provide rules on how to compose VOQL* basic formulas into *VOQL* well-formed formulas*, which we simply call *VOQL* formulas*.

3.2.1 VOQL* Basic Formulas

A VOQL* basic formula is a condition consisting of VOQL* simple terms and a comparison operator. Comparison operators may be represented by relative positions in space of VOQL* simple terms. Depending on the types of comparison operators, we distinguish VOQL* basic formulas into value-based conditions and oid-based conditions. Valued-based conditions and oid-based conditions are called *v-formulas* and *o-formulas*, respectively.



(a) (b) (c)

Fig. 4 Various forms of v-formulas. (a) the form of $t_1 \theta c$ (b) the form of $t_1 \theta t_2$ (c) the simpler form of $t_1 \theta t_2$

V-FORMULAS

V-formulas are atomic in the sense of atomic formulas in logic. V-formulas are of the form $t_1 \ \theta \ c$ or $t_1 \ \theta \ t_2$, where c is a constant in a primitive domain, and t_i a visual element or subblob. Generic forms of v-formulas are shown in Fig. 4 (a) and (b). Two VOQL* query expressions in Figs. 1 and 2 have three v-formulas of the form $t_1 \ \theta \ c$, corresponding to the following textual representation: Type =

"Corporation", City = "New York," and Age > 30. V-formulas of the form $t_1\ \theta\ t_2$ in Fig. 4 (b) are used to represent conditions corresponding to value-based joins in the relational model, where t_1 and t_2 are visual elements or subblobs in primitive domains.

We introduce the simpler form of $t_1\ \theta\ t_2$ in Fig. 4 (c) for notational simplicity. Using many visual constructs in a query may make it difficult to understand and decrease the benefit of using a visual query language.

O-FORMULAS

Borrowing the notations for set inclusion and set exclusion in Venn diagram, we spatially represent various conditions among elements, subblobs, and blobs. Simplest o-formulas among them consist of only two VOQL* simple terms. These formulas are called *atomic o-formulas*. Atomic o-formulas and their semantics are given in Fig. 5.

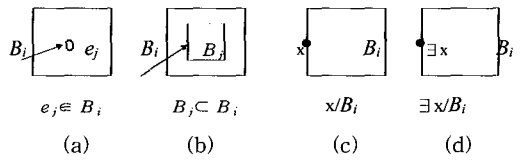


Fig. 5 Atomic o-formulas and their semantics.

In Fig. 5, the visual element, blob, subblob, and visual variable are labeled with e_j, B_i, B_j, and x, respectively, only for the purpose of defining the semantics of atomic o-formulas. These labels except the one for a blob will not be shown in ordinary VOQL* query expressions. Note that there are directed edges connected to the visual element e_j and the subblob B_j since they always exist as nodes of some VOQL* structured terms.

The semantics of o-formulas in cases (a) and (b) in Fig. 5 are straightforward. Case (c) in Fig. 5 shows the representation of a range clause in VOQL* and its semantics are the same as in OOPC. The o-formula in Case (d) represents the existentially quantified range clause The universal quantifier may be introduced as well. However, introducing the universal quantification to the language requires us to solve scoping problem among various variables, which is beyond the scope of this paper. Examples in Figs. 1 and 2 illustrate how these o-formulas are used in queries.

We can repetitively apply atomic o-formulas to produce more complex o-formulas and derive their semantics based on those of atomic o-formulas. However, there are formulas that simple inductive application of these atomic o-formulas cannot represent properly. These formulas are called *extended o-formulas*. We provide extended o-formulas and their semantic definition in Fig. 6.

Cases (a) - (c) in Fig. 6 represent o-formulas for conditions between subblobs at the same nesting level. Case (d) represents an o-formula for a condition between a nesting subblob and a nested subblob at multiple levels, and case (e) an o-formula for a condition of identity comparison between two visual elements. Case (f) shows the representation of a range clause based on a subblob and its semantics. The VOQL* query expression in Fig. 2 shows one range clause for case (c) in Fig.5 and two for case (f) in Fig. 6.

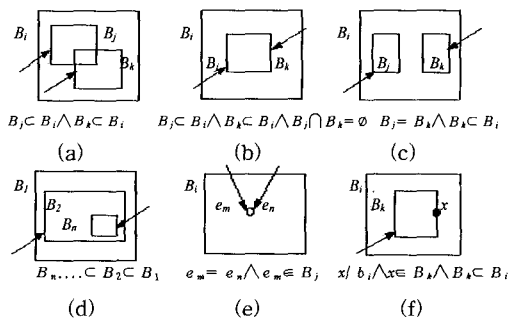


Fig. 6 Extended o-formulas and their semantics.

It may not be easy to define general syntactic rules (e.g. BNF for textual languages) capable of generating all o-formulas, due to the spatial nature of the language. Instead, we provided typical cases of o-formulas and their semantics in this section. We may use them as generic templates whenever we need. We may derive more useful and interesting o-formulas by adding additional visual constructs to

o-formulas in Fig. 5 and 6. We will further pursue this topic in subsequent papers.

3.2.2 VOQL* Well-Formed Formulas

In the previous section, we presented VOQL* basic formulas and their semantics. *VOQL* well-formed formulas*, simply called *VOQL* formulas*, can be constructed by creating various instances of the generic VOQL* basic formulas, combining and organizing them in certain ways. Different from the definition of wffs in logic, naive inductive application of VOQL* basic formulas does not produce a valid VOQL* formula. Note that visual elements and subblobs shown in VOQL* basic formulas cannot exist by themselves in a VOQL* formula, since they are meaningful only as nodes of VOQL* structured terms. A VOQL* formula is defined as a collection of VOQL* basic formulas satisfying the following conditions:

(C1) All the visual elements, visual variables, and subblobs shown in a VOQL* formula are nodes of some VOQL* structured terms. A visual element or a subblob may belong to more than one VOQL* structured terms, while a visual variable to only one VOQL* structured term.

(C2) Each visual variable, which is the root of a VOQL* structured term, participate in a range clause, as in Fig. 5 (c) and Fig. 6 (f). More than one visual variable may participate in a range clause.

(C3) All the leaf nodes of a VOQL* structured term participate in some VOQL* basic formulas. However, non-leaf nodes of path expressions may not.

(C4) Juxstaposition of more than one VOQL* formulas satisfying rules (C1)-(C3) above is also a VOQL* formula.

In order to illustrate (C1)-(C4), two VOQL* formulas are presented in Fig. 7 (a) and (b). (C1) says that no visual element or subblob can exist alone without being connected to a VOQL* structured term. (C2) implies that no visual variable can exist unless they participate in range clauses. (C3) means that if a leaf node of a VOQL structured term is not involved in any VOQL* basic formula, the whole VOQL* formula becomes invalid. (C4) means that the juxtaposition of more than one VOQL* formulas is interpreted as their conjunction.

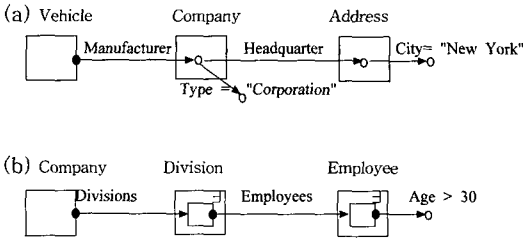


Fig. 7 Examples of VOQL* formulas

Before we present the algorithm converting a VOQL* formula into an OOPC wff, we introduce the notion of the *restricted relationship* between range clauses. For VOQL* structured terms vt_1 and vt_2, and a subblob sb_1 as a leaf node of vt_1, suppose that a range clause rc_i consists of the root of vt_1 and a blob (or a subblob), and that a range clause rc_k consists of the subblob sb_1 and the root of vt_2. Then, the range clause rc_k is defined to be *restricted by* the range clause rc_i. The VOQL* expression in Figs. 2 illustrates these restricted relationships among three range clauses.

The semantics of a VOQL* formula f is built in the following steps. First, we translate VOQL* simple terms into OOPC terms; then, for each individual basic formula in the formula f, produce an OOPC formula incrementally by applying the semantics of individual basic formulas defined in Fig. 4, 5, and 6. The order of applying the semantics of each basic formula in the formula f is important since various syntactic and semantic concepts in VOQL* that are represented in the spatial context should properly be transformed into one-dimensional text where the order of translated objects matters. A VOQL* formula f is translated into an OOPC wff by the algorithm *MakeFormula* as follows:

Algorithm MakeFormula

A. Get ready for the translation:

(1) Generate OOPC terms for class names of all

the blobs in *f*.

(2) Generate OOPC textual constants for textual constants in all the v-formulas in *f*.

(3) Assign distinct textual variables to all the visual variables.

(4) Generate OOPC path expressions for all the visual elements and subblobs in the VOQL* formula *f* as shown in Section 3.1.

B. Perform the translation

(1) Based on the semantic definition of VOQL* basic formulas in Section 3.1, choose a VOQL* basic formula in *f* and translate it into OOPC formulas using textual terms generated in Step A. Perform this step until no more basic formulas remain to process. Then, by removing all the AND connectives denoted by ∧ from the resulting OOPC formulas, generate a list of OOPC formulas without the AND connectives.

(2) Modify the order of OOPC formulas in the list resulting from step B.(1) in order to satisfy the following conditions: (a) all the range clauses and quantified range clauses are placed at the front of the list. (b) If a range clause rc_k is restricted by a range clause rc_i, the range clause rc_k should not be placed before a range clause rc_i.

(3) Remove commas from the list and connect all the OOPC formulas except all the range clauses in the list with the AND connective. Then, place the simbol [at the end of all the range clauses and the simbol] at the end of the resulting formula.

Suppose that we apply the algorithm MakeFormula to the VOQL* formula in Fig. 7. After applying the algorithm, we generated textual terms and an OOPC wff as follows:

Example in Fig. 7 (a)

Class names: Vehicle, Company, Address
Textual constants: "Corporation", "New York"
Path expressions:
x1, x1.Manufacturer, x1.Manufacturer.Type,
x1.Manufacturer.Headquarter, x1.Manufacturer.Headquarter.City

The list of OOPC Formulas converted from VOQL* basic formulas:
[x1/Vehicle, x1.Manufacturer ∈ Company, x1.Manufacturer.Type = "Corporation", x1.Manufacturer.Headquater ∈ Address, x1.Manufacturer.Headquater.City = "New York"]

The resulting OOPC formula:
x1/Vehicle [x1.Manufacturer ∈ Company ∧
x1.Manufacturer.Type = "Corporation" ∧
x1.Manufacturer.Headquater ∈ Address∧
x1.Manufacturer.Headquater.City = "New York"]

Example in Fig. 3.5 (b)

Classes names: Company, Division, Employee
Textual constants: 30
Path expressions : x1, x1.Divisions, x2, x2.Employees, x3, x3.Age

The list of OOPC Formulas converted from VOQL* basic formulas:
[x1/Company, ∃x2/Division, x2 ∈ x1.Divisions, x1.Divisions ⊂ Division, ∃x3/ Employee, x3 ∈ x2.Employees, *x2*.Employees ⊂ Employee, x3.Age > 30]

The resulting OOPC formula:
x1/Company ∃x2/Division ∃x3/ Employee [x2 ∈ x1.Divisions ∧ x3 ∈ x2.Employees ∧ x1.Divisions ⊂ Division ∧ *x2*.Employees ⊂ Employee ∧ x3.Age > 30]

3.3 VOQL* Query Expressions

A shaded stump blob connected to a visual variable or element with an undirected edge is called the *VOQL* target list*. Each shaded blob is labeled with the list of names of attributes to project. A VOQL* query expression consists of VOQL* target lists and a VOQL* formula. Since the syntax and semantics of VOQL* formulas have already been defined in Section 3.2, we focus on the syntax and semantics of VOQL* target lists in this section.

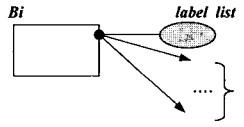
A VOQL* query expression (or simply a VOQL* expression) is defined by adding VOQL* target lists to a VOQL* formula. The semantics of VOQL* query expressions are given in Fig. 8. The expression in Fig. 8 (a) represents the case of a target list being directly connected to a visual variable; the one in Fig. 8 (b) the case of a target list being indirectly connected to a visual variable through visual elements. Let B_i denote a blob (or subblob) and *label list* a list of attributes. An empty list of labels in a target list implies returning all the attributes of the connected node (a visual variable or visual element) by default. Otherwise, it returns only the values of the attributes in the list.

For any VOQL* query expression *expr*, if we remove VOQL* target lists and undirected edges connected to them from *expr*, the resulting expression will be a VOQL* formula. Let the resulting formula

be *ufa*. The semantics of *expr* will be defined in the following steps:

1. Translate a VOQL* formula *ufa* into an OOPC wff using the algorithm MakeFormula. Let the resulting OOPC formula be *ofa*.
2. Generate an OOPC target list from a VOQL* target list of *expr* using textual terms generated in Step 1.
3. Find a range clause having a variable occuring in the OOPC target list from the OOPC wff *ofa* and remove the range clause from *ofa*. The resulting OOPC formula will be a qualification clause in OOPC. It is denoted as *qc* in Fig. 8. When the qualification clause *qc* is empty, it is vacuously true.
4. Generate OOPC range clauses, which are the clauses that have been removed in Step 3.

If no more VOQL* target lists remain to process, then stop. Otherwise goto Step 2.

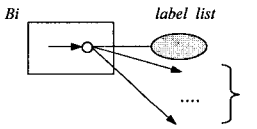


represent the rest of a VOQL* query expression consisting of 0 or more directed edges.

⇔ **if** empty(*label list*) **then** { *x* | *x/Bi*; *qc* }
else{*x.label1*, ···, *x.labeln* | *x/Bi*; *qc* }

where *x* is the visual variable in *Bi*, *label list* is *label1*, ···, *labeln*, and ∃*x/Bi qc* represents the OOPC wff translated from the rest of the VOQL* query expression.

(a)



represent the rest of VOQL* query expression consisting of 0 or more directed edges.

⇔ **if** empty(*label list*) **then** { *x.att1*···*attk* | *x/Bi*; *qc* }
else{*x.att1*···*attk.label1*, ..., *x.att1*···*attk.labeln* | *x/Bi*; *qc* }

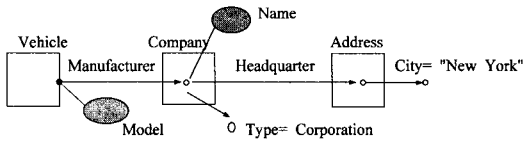
where *x.att1*···*attk* is the OOPC path expression of the element in *Bi*, *label list* is *label1*···, *labeln*, and ∃*x/Bi qc* represents the OOPC wff translated from the rest of the VOQL* query expression.

(b)

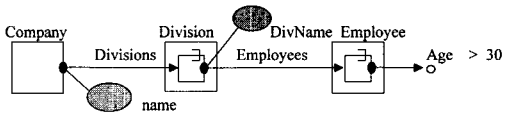
Fig. 8 VOQL* query expressions and their semantics

Multiple VOQL* target lists may exist for one VOQL* query expression. Multiple VOQL* target lists in a query expression are represented by Cartesian Product of lists of attributes. We are not concerned about the order of target lists in this paper.

Consider the example in Fig. 9 (a) and (b). The VOQL* query expressions are translated into the OOPC expressions in Fig. 9 (c) and (d).



(a)



(b)

{*x1*.Model, x1.Manufacturer.Name | *x1*/Vehicle; *x1*.Manufacturer ∈ Company ∧ *x1*.Manufacturer.Type = "Corporation" ∧ x1.Manufacturer.Headquater ∈ Address ∧ *x1*.Manufacturer.Headquarter.City = "New York"}

(c)

{*x*1.Name, *x2*.DivName| *x*1/Company, *x2*/Division; ∃*x3*/Employee [*x1*.Divisions ⊂ Division ∧ *x2* ∈ *x1*.Divisions ∧ *x2*.Employees ⊂ Employee ∧ *x3* ∈ *x2*.Employees ∧ *x3*.Age > 30]}

(d)

Fig. 9 Examples of VOQL* query expressions

4. More VOQL* Query Examples

In this section, we present more query examples in order to illustrate the intuitiveness and usefulness of VOQL* as a visual query language. Consider the following queries:

(a) "Retrieve the vehicles such that the headquarter of the manufacturer and the dealer are located in the same city."

(b) "Retrieve the vehicles such that the headquarter of the manufacturer and the dealer are located in the city of New York, and moreover, at the same address."

(c) "Retrieve the name of a division in GM and the name of an employee working in the division such that all the vehicles the employee owns are assmebled by the division."

(d) "Retrieve the name of a division in GM and the name of an employee working in the division such that none of the vehicles the employee owns is assmebled by the division."

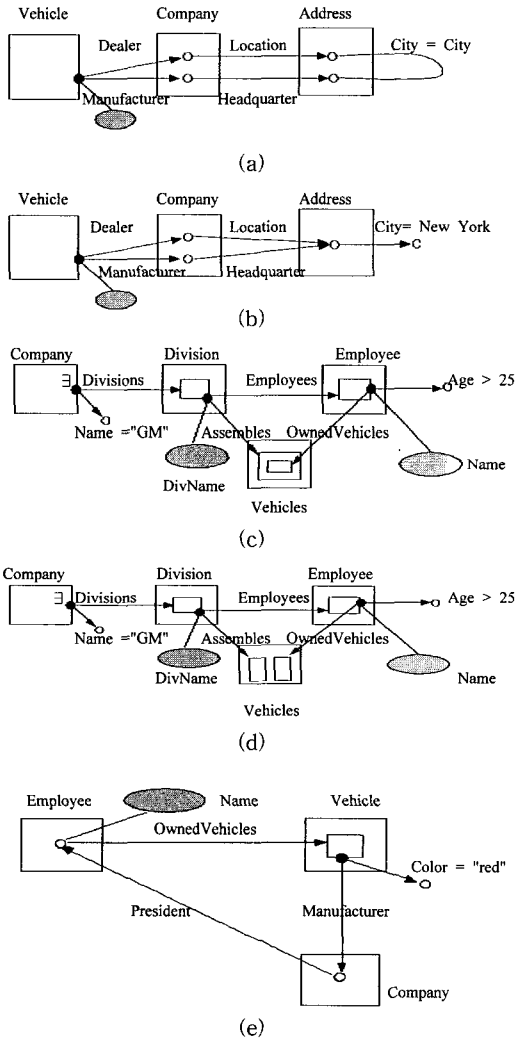


Fig. 10 Examples of VOQL* query expressions.

(e) "Retrieve the name of an employee, (1) who has a vehicle with red color (2) and the manufacturer of which is the company where he is the president of."

VOQL* representations for the queries above are given in Fig. 10. Note that a VOQL* basic formula (a v-formula) of the form (c) in Fig. 4 is used in Fig. 10 (a). The VOQL* query in Fig. 10 (b) uses a o-formula of the generic form in Fig. 6 (e). VOQL* queries in Fig. 10 (a) and (b) are translated into the following OOPC queries.

(a) { x | x/Vehicle; x.Dealer ∈ Company ∧ x.Manufacturer ∈ Company ∧ x.Dealer.Location ∈ Address ∧ x.Manufacturer.Headquarter ∈ Address ∧ x.Dealer.Location.City = x.Manufacturer.Headquarter.City };

(b) { x | x/Vehicle; x.Dealer ∈ Company ∧ x.Manufacturer ∈ Company ∧ x.Dealer.Location ∈ Address ∧ x.Manufacturer.Headquarter ∈ Address ∧ x.Dealer.Location = x.Manufacturer.Headquarter ∧ x.Dealer.Location = "New York" };

VOQL* queries for queries (c), (d), and (e) are represented in Fig. 10 (c), (d), and (e). Note that the query in Fig. 10 (c) uses an o-formula of the form in Fig. 6 (d), while the one in Fig. 10 (d) an o-formula of the form in Fig. 6 (c). Note that the query (e) is cyclic. Their OOPC translation is as follows.

(c) { $x2$.DivName, $x3$.Name | $x2$/Division, $x3$/Employee ; ∃$x1$/Company [$x1$.Name = "GM" ∧ $x2$ ∈ $x1$.Divisions ∧ $x1$.Divisions ⊂ Division ∧ $x3$ ∈ $x2$.Employees ∧ $x2$.Employees ⊂ Employee ∧ $x2$.Assembles ⊂ Vehicles ∧ $x2$.Assembles ⊃ $x3$.OwnedVehicles ∧ $x3$.Age > 25] };

(d) { $x2$.DivName, $x3$.Name | $x2$/Division, $x3$/Employee ; ∃$x1$/Company [$x1$.Name = "GM" ∧ $x2$ ∈ $x1$.Divisions ∧ $x1$.Divisions ⊂ Division ∧ $x3$ ∈ $x2$.Employees ∧ $x2$.Employees ⊂ Employee ∧ Vehicles ⊃ $x2$.Assembles ∧ Vehicles ⊃ $x3$.OwnedVehicles ∧ ($x2$.Assembles ∩ $x3$.OwnedVehicles) = Ø ∧ $x3$.Age > 25] };

(e) { x.Manufacturer.President.Name | x/Vehicle; x ∈ x.Manufacturer.President.OwnedVehicles ∧ x.Manufacturer.President.OwnedVehicles ⊂ Vehicle ∧ x.Color = "red" ∧ x.Manufacturer ∈ Company ∧ x.Manufacturer.President ∈ Employee }

Since the translation process of queries in Fig.10 (a)-(d) is rather straightforward, let us focus how the query in Fig.10 (e) is translated. By applying Algorithm MakeFormula to the query, we can generate path expressions such as { x, x.Color, x.Manufacturer, x.Manufacturer.President, x.Manufacturer.President.OwnedVehicles }. The query expression has four VOQL* basic formulas. Let us illustrate how these formulas are translated into corresponding OOPC wffs as follows:

(1) one VOQL* basic formula for Case (f) in Fig. 6:
x/Vehicle ∧ x ∈ x.Manufacturer.President.-OwnedVehicles ∧ x.Manufacturer.President.-OwnedVehicles ⊂ Vehicle

(2) one VOQL* basic formula for Case (a) in Fig.4:
Color = "red"

(3) two VOQL* basic formulas for Case (a) in Fig. 5
x.Manufacturer ∈ Company,
x.Manufacturer.President ∈ Employee

After applying Step B of Algorithm MakeFormula, we have the following formula:

x/Vehicle [*x* ∈ *x*.Manufacturer.President.OwnedVehicles ∧ *x*.Color = "red" ∧ *x*.Manufacturer ∈ Company ∧ *x*.Manufacturer.President ∈ Employee]

According to the semantics of VOQL* query expressions, we can easily derive the OOPC query expression (e) from the formula above.

4. Closing Remarks and Future Research

The contributions of VOQL* may be summarized as the excellent expressive capability of strucutured objects and set-related conditions, the syntax reflecting the syntactic structure of OOPC, and the inductively defined formal semantics based on OOPC. The language constructs of VOQL* are based on Higraph. In Higraph, the notion of sets is represented with the Venn diagram, while that of nested structures with the graph. The language constructs corresponding to the Venn diagram and the graph in VOQL* enable us to naturally represent both set-related conditions and tree-strucutured terms.

As textual object query languages, XSQL [12] and PathLog [9] have proposed text-based strucutured path expressions as an extension to simple linear path expressions. However, their path expressions represented by adding another dimension to linear path expressions are not as intuitive nor as powerful as VOQL* structured terms.

Recently, two visual object query languages such as QUIVER [6] and PESTO [20] have been proposed. QUIVER has the language constructs similar to blobs, edges and visual elements. With these graphical language constructs, QUIVER has better representation of structured path expressions. However, QUIVER does not have a formal semantics. It also fails to support set-related conditions such as o-formulas in VOQL*, and structured path expressions such as VOQL* structured terms.

PESTO may be viewed as a browsing system equipped with querying capability based on QBE. PESTO provides a nice user interface as in many other visual query languages based on QBE. However, PESTO lacks a formal semantics. Its expressive capability of representing structural relationships among objects and various set-related conditions falls short of VOQL* and QUIVER.

Wed like to comment on the user interface of a querying system implementing VOQL*. We believe that the user interface of QUIVER is better than that of PESTO in representing structural relationships, while PESTO provides better user interface in specifying conditions than QUIVER does. We believe that the user interface of a querying system for VOQL* should be a hybrid approach of the graphical approach and the approach based on QBE.

The syntax of VOQL* has been carefully designed to visually simulate the syntactic strucuture of logic, defined on the notions of variables, constants, terms, and formulas. Due to these characteristics, the syntax of VOQL* is intuitive and comprehensible; the semantics of VOQL* are clear, concise, and most importantly, formally defined by induction. We believe that no other visual query language has formal semantics inductively defined based on logic.

Even though VOQL* has many good features, there are many things to be done. In this paper, we have not specified how to represent projection list and aggregation operators. We have been defining and implementing them in the interpretor for VOQL*, which we have been constructing. The results on these issues will be presented soon in forthcoming papers. The current version of VOQL* is conjunctive. We are also working on the design of operators such as negation, disjunction, implication, and also on the scoping problem. We believe that the well-defined syntax and sematics of VOQL* may enable us to extend the VOQL* without much difficulty.

References

[1] Angelaccio, M., Catarci, T., and Santucci, G., "QBD*: A Graphical Query Language With Recursion," *IEEE Trans. on Software Engineering*, Vol. 16, No. 10, pages 1150-1163, October 1990.

[2] Bancilhon, F., Delobel, C., and Kanellakis, P., *Building an Object-Oriented Database System, The Story of O2*, Morgan Kaufmann, San Mateo, CA, 1992.

[3] Beeri, C., "Formal Models for Object-Oriented Databases," In *Proc. 1st Intl Conf. on Deductive and Object-Oriented Databases*, pages 370-395, Kyoto, Dec. 1989.

[4] Bertino, E. et al., "Object-Oriented Query Languages: The Notion and the Issues," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 1, No. 3, pages 223-237, June 1992.

[5] Cattell, R.G.G et al., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, San Francisco, CA, 1997.

[6] Chavda, M. and Wood, P., "Towards an ODMG-Compliant Visual Object Query Language," In *Proc. the 23rd Intl Conf. on Very Large Data Bases*, pages 456-465, Athens, Greece, 1997.

[7] Cruz, I., Mendelzon, A., and Wood, P., "Graphical Query Language Supporting Recursion," In *Proc. Intl Conf. on Management of Data, ACM SIGMOD*, pages 323-330, 1987.

[8] Czejdo, B., Elmasri, R., and Rusinkiewicz., M., "A Graphical Data Manipulation Language for an Extended Entity-Relationship Model," *IEEE Computer*, Vol.23 pages 26-36, Mar.1990.

[9] Frohn, J., Lausen, G., and Uphoff, H., "Access to Objects by Path Expressions and Rules," In *Proc. the 20th Intl Conf. on Very Large Data Bases*, pages 273-284, 1994.

[10] Goldman, K.J., Goldman, S.A., Kanellakis, P.C., and Zdonik, S.B., "ISIS: Interface for a Semantic Information System," In *Proc. Intl Conf. on Management of Data, ACM SIGMOD*, pages 328-342, May 1985.

[11] Gyssens, M. et al., "A Graph- Oriented Object Database Model," *IEEE Trans. on Knowledge and Data Engineering*, Vol.6, No.4, pages 572-586, 1994.

[12] Kifer, M., Kim, W., and Sagiv, Y., "Querying Object-Oriented Databases," In *Proc. Intl Conf. on Management of Data, ACM SIGMOD*, pages 393-402, San Diego, CA, 1992.

[13] Kim, J.H., Han, T.S., and Lee, S.K., VOQL: A Visual Object-Oriented Database Query Language For Visualizing Path Expressions, *Computer Systems, Science and Engineering*, accepted to appear.

[14] Kim, J.H., Han, T.S., and Lee, S.K., Visualization of Path Expressions in a Visual Object-Oriented Database Query Language, In *Proc. Intl Conf. on Database Systems for Advanced Applications*, page 99-108, Taiwan, 1999.

[15] Kim, W., *Introduction to Object-Oriented Databases*, The MIT Press, 1990.

[16] Mohan, L. and Kashyap, R. L., "A Visual Query Language for Graphical Interaction With Schema-Intensive Databases," *IEEE Trans. on Knowledge and Data Engineering*, Vol.5, No.5, pages 843-858, 1993.

[17] Mylopoulos, J., Bernstein, P. A., and Wong, H. K. T., "A Language Facility for Designing Database-Intensive Applications," *ACM Trans. on Database Systems*, Vol.5, No.2, pages 185-207, 1980.

[18] Sockut, G. H., Burns, L. M., Malhotra, A., and Whang, K-Y., "GRAQULA: A Graphical Query Language for Entity-Relationship or Relational Databases," *Data and Knowledge Engineering*, Vol.11, pages 171-202, 1993.

[19] Vadaparty, K., Aslandogan, Y. A., and Ozsoyoglu, G., "Towards a Unified Visual Database Access," In *Proc. Intl Conf. on Management of Data, ACM SIGMOD*, pages 357-366, 1993.

[20] Carey, M., Haas, L., Maganty, V., and Williams, J. PESTO: An Integrated Query/Browser for Object Databases, In *Proc. the 22th Intl Conf. on Very Large Data Bases*, pages 203-214, 1996.

이 석 균

1982년 서울대학교 경제학 학사. 1990년 University of Iowa 전산과학 석사. 1993년 University of Iowa 전산과학 박사. 1992년 IEEE 8th International Conference on Data Engineering에서 최우수 논문상 수상. 1993년 9월 ~ 1997년 2월 세종대학교 정보처리학과 전임강사. 1997년 ~ 현재 단국대학교 전산통계학과 조교수. 관심분야는 데이타 모델링, 데이타베이스에서 불완전정보관리, 객체지향, 데이타베이스 시스템, 데이타베이스 질의어, 다중처리기에서의 실시간 스케줄링, 데이타웨어하우스