International Journal of Computer Science & Applications
Vol. 5, No. 2, pp 23 - 51

THE GOQL LANGUAGE AND ITS FORMAL SPECIFICATIONS

Euclid Keramopoulos, Philippos Pouyioutas , Tasos Ptohos
INTERCOLLEGE
46 Makedonitissas Avenue, P.O. Box 24005
1700 Nicosia , Cyprus
Email: pouyioutas.p@intercollege.ac.cy

Abstract

The Graphical Object Query Language (GOQL) is a graphical query language that complies with the ODMG standard and runs on top of the o2 DBMS. The language provides users with the User's View (UV) and the Folders Window (FW), which serve as the foundation upon which end-users can pose ad-hoc queries. The UV is a graphical representation of any underlying ODMG scheme. Among its advantages is that it hides from end-users most of the perplexing details of the object-oriented database model, such as methods, hierarchies and relationships. To achieve this, the UV does not distinguish between methods, attributes and relationships, it encapsulates is-a hierarchies and it utilises a number of desktop metaphors whose semantics can be easily understood by end-users. The FW is a condensed version of the UV and provides the starting point for constructing queries. In this paper, we demonstrate, using an example, the UV and the FW and the way they support the construction of graphical queries. We then present the formal specifications of the language. We first give a formal definition of an object-oriented database schema in the GOQL model. The UV is then formally defined as a mapping from a GOQL object-oriented database schema. The formal definition of the UV allows us to formally define the graphical constructs of GOQL and the syntax analysis of the language.

Keywords: Graphical Query Languages, Query Language, OODBM, Formal Specifications

1. Introduction

Following the introduction of Graphical User Interfaces (GUIs) by Apple and Microsoft the use of graphical interfaces has become fashionable. A number of graphical programming languages have been introduced. In the database community the importance of graphical representation to conceptual design was recognised early and research into this field led to the development of a number of semantic models such as the Entity Relationship Model [1]. Recently, there was considerable interest in the development of Graphical Query Languages (GQLs) [2-12]. GQLs are graphical user interfaces that allow users to query (retrieve, create, delete and update) their underlying database, i.e. GQLs are special purpose graphical user interfaces.

Most of the existing GQLs are not addressed to naive users because of the way they represent graphical schemes. In particular, most of the GQLs either do not support all the features of the underlying Database Model (DBM) or, if they do support them, they end up representing perplexing (for the non-expert user) features (e.g. inheritance, types, behaviour, etc.) of the underlying DBM. Moreover, most of these GQLs use mathematical symbols for the representation of certain features (e.g. logical operators), which according to the results of our investigation, can be used only by users with a basic and/or sometimes good mathematical background. We believe the main reason for this, is that either these languages have been designed to address users' needs or groups of both naive and expert users have not been used to evaluate sufficiently the various (if any) design human factor artefacts (metaphors, colour) employed.

Moreover, GQLs employ a graphical scheme to represent the structures of the underlying DBM, which are illustrated by a transformation of all the features of the underlying database scheme into graphs. Also, as the vast majority of GQLs set out the design of a query from this graphical scheme, this graphical scheme imposes restrictions on the GQL depending on the DBM employed.

In agreement with [7], we consider that another drawback of most of the existing GQLs is their poor formal definition. In [7], it is claimed that a GQL is well defined when it is defined either on a mathematical language like Z [13], VDM [14] or when it is defined on another language, which is well-defined. However, the issue of formal definition has been addressed only by a handful of GQLs found in literature [2, 5, 7, 8]. Finally, the expressive power of a language needs to be proved mathematically; again only a small proportion of the GQLs found in the literature prove their expressive power.

Our research on graphical query languages has contributed to the subject area as follows. First of all, we designed a new graphical scheme representation, namely *User's View*, which has as basic characteristics the use of (a) human factors, like desktop metaphors and colour, and (b) the elimination of technical details without loosing anything of the database model power. The contribution of our research is that the User's View is addressed to all types of users including naive ones and also, that it is designed to be independent of the underlying database model.

Besides that, we designed a new graphical query language, namely *GOQL* (Graphical Object Query Language), which has the same expressive power as the ODMG 3.0 standard OQL [15] and it is the only GQL for the ODMG 3.0 that supports also binding functions and method parameters. Moreover, we showed that the use of colour and non-abstract representation result in high understandability by organising a formal experiment. We based on that the design of GOQL, where we also used metaphors and colour. Finally, we evaluated the User's View and GOQL with all types of end-users and we formally defined and implemented them [16].

In this paper, in Section 2 we discuss the principles and characteristics of Graphical Query Languages. In Section 3, we demonstrate, using an example, the UV and the FW and the way they support graphical queries. In Section 4, we explain the design of the GOQL language and the concepts considered during the design stages. In section 5, we explain the User Interface of GOQL and illustrate the use of the language by giving sample queries. In Section 6, we present briefly the GOQL system architecture and address some implementation issues In Section 7, we provide a formal definition of the GOQL object-oriented database model. In Section 8, we formally define the mapping from a GOQL object-oriented database schema to its corresponding UV. This mapping is used to construct the UV of a given object-oriented database schema. Finally, the paper concludes by discussing our current and future work.

2. Principles and Characteristics of Graphical Query Languages

In this section, we present the features of an analysis methodology that is employed to evaluate/characterise a graphical query language and present a critical evaluation of GOQL as compared to the other existing graphical query languages. Our analysis methodology has adopted some characteristics from the analysis methodology on query languages proposed in [17] and from the survey on graphical query languages on databases of [18]. A detailed analysis of our research work presenting a comparison analysis of the existing graphical query languages based on the proposed methodology can be found in [19]. The analysis includes amongst others the languages and models of ODMG 3.0 [15], UML [20,21], COAD/YOURDON [22], ERM [1] and EERM [23], AMAZE [4], G-Log [8], GOMI [3], Khoshafian Model [24], Kaleidoquery [6,25], PICASSO [9], Pasta-3 [10], QBD* [2, 26], SUPER [11], Gql [7, 27], OdeView [12], QUIVER [5, 28], MS-ACCESS [29], GQL [30], Business Objects [31] and Oracle [32].

In particular, the following features comprise our analysis methodology:

- The level of **expertise that users** of a GQL should have.
- The **underlying data model** supported by the graphical query language because it is the underlying data model that determines the querying mechanism.
- The way a **graphical scheme** is utilised in formulating a query.
- The **design characteristics**, which include functions that are supported by the query system, a language form and the expressivity of the language.
- The existence of a **formal definition**, i.e. whether a syntactic analysis, a semantic analysis and a proof of the expressive power have been proposed.
- The platform(s) used for the **implementation** of the language.
- The **evaluation method** used to measure the quality of a GQL.
- The support of any **human factors**, such as metaphors and colour.
- The way a **query output** is presented.

In Table 1 and Table 2 below, we present the results of our evaluation. Table 1 presents a comparison analysis of features supported by the graphical schemes of various graphical query languages, whereas Table 2 presents the functions supported by the various graphical languages. In the rest of this section we discuss the results of our evaluation and address the advantages of our query language GOQL.

Graphical Scheme / Features	ODMG 3	COAD/YOURDON	EER	AMAZE	G-Log	GOMI	Khoshafian	Kaleidoquery	UML	PICASSO	Pasta-3	QBD*	SUPER	Gql
class/entities	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
attributes		✓	✓	✓	✓	✓	✓		✓	✓			✓	✓
methods		✓			✓		✓		✓					
method parameters														
relationships	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
hierarchy	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	
recursion	✓	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓	✓
complex properties		✓	✓		✓	✓	✓						✓	✓
collection properties							✓							
metaphors				✓				✓						

Table 1: Features of Graphical Schemes

2.1 User Type

Most of the GQLs are designed for skilled and expert users, we perceive this as an indication that designers tend to sacrifice simplicity/understandability to achieve a richer expressivity. For example, GOMI [3] has been created for users with knowledge of object-oriented technology; thus, the language elaborately employs an object-oriented scheme with which its target users are familiar with; AMAZE [4], on the other hand, has been designed having naive users in mind, so its designers have utilised an interface and a notation which they believe it would appeal to their target users.

2.2 Data Model

Each of the presented graphical query languages is associated with a particular underlying DBM and the language's design reflects the functionality of that underlying data model. Thus, users can work using only a relational database system, a functional database system, an object-oriented database system or an enhanced ERC+ in case of SUPER [11]. Furthermore, the vast majority of the graphical query languages, e.g. Gql [7], GOMI [3], use the scheme of the underlying database as the starting point for the queries' construction. The implications of these are numerous. Firstly, none of the presented graphical query languages contributes anything towards bridging the gap that exists between the various data models. Secondly, the limitations of each GQL's underlying data model are inherited and reflected by the GQL itself. Furthermore, queries which are expressed in a graphical query language have to be mapped to queries of the underlying data model, which means that the expressive power of a graphical query language is frequently the same as that of the underlying model. Finally, because graphical query languages are based on a specific data model they impose further problems to users, i.e. users have to understand the underlying data model in order to have a better grasp of the workings of the graphical query language they are using; thus, if the underlying database is to be changed, then almost definitely the graphical query languages have also to be scrapped.

2.3 Graphical Scheme Representation

The query construction mechanism employed by the majority of the evaluated GQLs has as a starting point the database scheme. Thus, a GQL may be considered as a textually query language when it does not represent graphically the database scheme. Moreover, the absence of a graphical scheme from a GQL reduces the level of simplicity from a language, because the textually presentation of the database scheme includes some perplexing details especially for naive users. Most of the graphical schemes (see Table 2) do not represent all the features of the underlying database scheme graphically. For example collections are supported only in Khoshafian [24]; GOMI [3] supports collections in a textual way, i.e. the collection type is written next to the attribute. Also, we have not found in the literature any graphical representation of the method parameters. Finally, Kaleidoquery [6] and AMAZE [4] are the only ones that use metaphors.

Features	GQLs	PICASSO	Pasta-3	QBD*	G-Log	AMAZE	SUPER	Ode View	Gql	GOMI	Kaleidoquery	Quiver	Access
User Type	Naive					✓			✓			✓	
	Skilled	✓	✓	✓				✓		✓	✓		✓
	Expert				✓		✓						
Data Model	Relational	✓	✓	✓			✓						✓
	Object-Oriented				✓	✓		✓		✓	✓	✓	
	Functional								✓				
Graphical Scheme		✓	✓	✓	✓	✓	✓		✓	✓	✓		✓
Understanding the reality of interest	Top-Down		✓				✓						
	Browsing				✓				✓	✓	✓	✓	
	Scheme Simplification			✓									
Functionality	Retrieval	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Update		✓				✓	✓		✓			✓
	Organisational		✓				✓	✓		✓			✓
Language Form	2D	✓	✓	✓	✓		✓		✓	✓		✓	✓
	3D					✓					✓		
	Textual							✓					
	Form-based							✓					✓
	Diagram-based	✓	✓	✓	✓	✓	✓		✓			✓	
	Icon-based												
	Hybrid									✓	✓		
Expressivity	Predicates	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Boolean & Set Operators	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Arithmetic Expressions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Existential/Universal Quantifiers	✓	✓		✓		✓		✓	✓	✓		✓
	Aggregate Functions	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Group-by Operator	✓	✓	✓	✓				✓		✓		✓
	Binding Functions		✓						✓				✓
	Recursion		✓	✓	✓					✓			
Methods	Yes							✓			✓	✓	
	Not applicable	✓	✓	✓			✓		✓				✓
Formal Definition	Syntactic Analysis			✓	✓				✓			✓	
	Semantic Analysis			✓	✓				✓			✓	
Implementation		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Evaluation				✓							✓	✓	
Human Factors	Colourful				✓	✓					✓	✓	
	Metaphors					✓					✓		
Output	2D	✓		✓				✓		✓			✓
	3D					✓							
	Textual		✓		✓		✓		✓		✓	✓	

Table 2: Functions of Graphical Languages

2.4 Understanding the Reality of Interest

To construct a query, the majority of GQLs use as a starting point the graphical database scheme. Some of the presented GQLs (like AMAZE [4] and PICASSO [9]) modify the DBM scheme to reflect constraints imposed by a query; this overloads the scheme with information, which can be overwhelming. OdeView [12] constructs queries using entity/class properties hiding at the same time any relationships and paths that may exist between these entities/classes; this gives a rather textual syntax to their queries. Pasta-3 [10] and SUPER [11] use a top-down method where the graphical scheme is reduced by eliminating any non-relevant, to the query, entities/classes. QBD* [2] supports scheme simplification, which reduces and renames some parts of the initial scheme resulting to a simplification of the scheme in a way that it will be a closer representation of the query demands. Finally, most of the discussed GQLs provide a browsing query construction mechanism, where the user navigates in a database scheme from one element to another element.

2.5 Functionality

All the query languages we have examined support a retrieval function. Only five of the examined GQLs namely, Pasta-3 [10], SUPER [11], OdeView [12], GOMI [3] and Microsoft Access [29] support an update

function. Of the above five GQLs, SUPER [25] provides only a data definition mechanism, whereas the remaining four GQLs support also an access control mechanism.

2.6 Language Form

Most of the surveyed languages support a 2D interface, which we believe is the most appropriate interface for a GQL. According to [7] interfaces that utilise a textual syntax to represent queries within a graphical environment are not user friendly. On the other hand, 3D interfaces are depending on the devices used to display/view graphics and can be confusing for users, especially when they are used with a database scheme that contains lots of structures.There are four types of interfaces, namely diagram-based interfaces, form-based interfaces, icon-based interfaces and hybrid ones. The majority of GQLs use a diagram-based interface. Microsoft Access [29] is an example of a form-based interface, whereas Kaleidoquery [6] provides a hybrid-based interface, which is mainly diagram-based but it also utilises some icon features. We believe that the most appropriate interface is a hybrid-based one as it combines the advantages offered by the other three interface types to achieve the simplest and user-friendliest presentation of data and query design.

2.7 Expressivity and Methods

From the presented GQLs, only Pasta-3 [10] supports all the expressive functions. Most of the graphical object-oriented query languages support only class hierarchies but OdeView [12] and QUIVER [5, 28] also support methods. The former has a textual structural appearance, whereas the latter has a very poor expressive power. Mathematical symbols are used to express cardinalities and quantification in most of the existing graphical query languages. Finally, binding functions is a feature of the ODMG 3.0 standard [15], which is not supported by any GQL defined for the Object-Oriented Data Model.

2.8 Evaluation

From the presented GQLs only three have been evaluated. QBD* [2] was compared against SQL [33] and QBI [34] by undergraduate students, secretaries and professionals [18]. It was found easier than SQL [33] and equally easy to QBI. Actually, QBD* was easier for expert users but more difficult for the naives and also for different types of queries was easier either QBD* or QBI. Kaleidoquery [6] was compared against OQL and found easier for the naive users and equal easy for the experts. Finally, QUIVER [5, 28] was compared against OQL [15] and found easier than OQL.

2.9 Formal Definition

Only four of the presented GQLs have been defined formally. One of the attempts to formally define a GQL was given in Gql [7], using mathematical sets and functions. The syntactic definition was consisted of the definition of the graphical scheme, of all the graphical and non-graphical constructs, and was completed by visualisation and syntax rules. The Gql's operational semantics were defined by translating the Gql's graphical instances into FDL list comprehension. Another GQL formally defined is QBD* [2], where a mapping from the graphical operations to a textual language is described and the syntax of the textual language is given in BNF format. In G-Log [8] the syntax is given by a number of definitions, which describe the valid rules of the language. Finally, in QUIVER [5, 28] the syntax of the language is defined by context-free grammar, whereas the operational semantics are given by translating QUIVER [5, 28] graphical queries into OQL statements. We believe that the expressive power can be defined only for those GQLs that have been defined formally, i.e. the above four GQLs (Gql, QBD*, G-Log and QUIVER).

2.10 Implementation

All the surveyed GQLs have been implemented at least as a prototype. All of them consisted at least of a graphical user interface and a translator that was used to transform the graphical query statements into equivalent query statements of an existing query language. Some of the GQLs (like Gql, AMAZE, QBD*, Pasta-3) provide also a database scheme presenter to present graphically the database scheme. Finally, few GQLs (like GOMI, OdeView, AMAZE) supported an output presenter, which presents the results of a query statement in a graphical way.

2.11 Human Factors

Most of the presented GQLs do not incorporate human factors in their design. Colour is supported only by four of the discussed GQL, whereas metaphors are utilised by AMAZE [4] and Kaleidoquery [6]. AMAZE [4] uses the 'castle' metaphor, which places subclasses on top of its superclass in order to give a better presentation of a hierarchy. Kaleidoquery [6] uses different icons as metaphors to represent classes. However, we do not think that this is the most suitable way for a scheme representation. Also, Kaleidoquery uses the 'water flow' metaphor for presenting graphical query syntax.

2.12 Output Representation

The majority of GQLs provide a simplistic textual, tabular or non-tabular representation for the query results. GOMI [3] takes a diagrammatic approach to the presentation of query results. AMAZE [4] also outputs query results in 3D form and OdeView [12] presents every object of the query output in a different window. We believe that GOMI's, AMAZE's and OdeView's approach can be too complicated especially when a response to a query generates lots of data that need to be presented, as a large number of output results will generate a non-readable representation of the data involved.

2.13 GOQL Features

Based on all the above findings, our research has been focused on the design, evaluation, formal definition and implementation of a graphical query language. The data model chosen to be used as the underlying database model in GOQL language is the ODMG 3.0 standard [15]. The language's design involved the design of graphical representation of the underlying database scheme. This graphical representation, which is called the *User's View*, is designed to be independent of concepts related to the underlying database model. In particular, GOQL is designed to address the needs of all the types of users, including the naive one. To do this, the language is designed to **eliminate perplexing technical details**, that can alienate users, without loosing anything of the database expressive power; this means that GOQL supports **method parameters** passing something that none of the other GQLs does. Moreover, a set of **desktop metaphors** is utilized to represent database characteristics to make complex concepts easier to understand.

The basic characteristics of the proposed GQL are the following:

- GOQL is **based on ODBMS.**
- It supports only **retrieval functionality.**
- The language's form is **two dimensional** and **hybrid**, i.e. the GOQL design uses forms, diagrams and icons.
- The **expressivity** of GOQL is the **same as ODMG 3.0 OQL** [15]. Moreover, GOQL supports **binding functions** and **method parameters**, a feature that is not supported by any other GQL based on the ODBMS.
- **The understandability of basic features of GOQL is evaluated at the design stage using a formal experiment.** In particular, the formal experiment is organized to check the hypothesis that "*the use of colour and non-abstract representation can result in high understandability*".
- **GOQL is formally defined** in two phases, namely the mapping from the ODBMS (data model) to the User's View, and the syntax and semantics of the language.
- GOQL was **implemented** as a prototype using Tcl/Tk.
- For the design of GOQL two human factor aspects are used, namely **desktop metaphors** and **colour**.
- The output presentation of GOQL is in an ODMG 3.0 OQL [15] statement.

3. The User's View (UV) and the Folders Window (FW) of GOQL

GOQL was designed to address the needs of end-users and to provide an alternative graphical query language to OQL. Thus, GOQL was designed to comply fully with the features of the object model of the ODMG 3.0 [15] and its query language, OQL. GOQL allows users to express graphically a variety of ad hoc queries ranging from simplistic ones to rather complicated ones. The features provided/supported by GOQL include: the support of a 2D colour interface, the use/support of methods, predicates, Boolean & set operators, arithmetic expressions, existential /universal quantifiers, aggregate functions, group by and sort operators,

functions, and sub queries. To achieve these, GOQL users are presented with the User's View (UV), which is GOQL's graphical representation of an underlying ODMG database scheme and which serves as the foundation upon which GOQL queries are constructed. In this section we address the importance of graphical scheme representations and the use of metaphors in constructing these representations. We also provide the running example of the paper (see Appendix I for OQL code).

3.1 Graphical Scheme Representations

The importance of graphical scheme representations of database constructs has been recognised in the late 70s following the proposal and success of the entity/relationship (E/R) model [1]. Since then graphical scheme representation has been used for the definition of data models (EERM [23]) and even for the representation of data in languages (UML [20, 21, 30]). The main objective of graphical scheme representation is to provide a simple and user-friendly alternative to the way database structures are conceptualised.

Graphical schemes are constructed using the stored metadata about database schemes. Their design is such that it emphasises certain important characteristics of the database scheme they represent. It is obvious that the complexity and detail, in which characteristics of a database scheme are represented by a graphical scheme is important for the target group of users. In particular, skilled and expert users should be in a position to work with a graphical scheme designed for naive users, whereas a graphical scheme designed for an expert user may be difficult to be understood by skilled and/or naive users.

Most of the graphical schemes found in the literature with the exception of Kaleidoquery [6] and AMAZE [4] are addressed to expert users. This is because metaphors are not used in the graphical representation of these schemes. Moreover, all proposed schemes, represent graphically all the technical details of the underlying database model without trying to hide/metaphorically present some perplexing details that confuse non-expert users.

3.2 Desktop Metaphors

Quite frequently, in everyday life, attempts to explain or simplify something involve employing examples that the target audience can relate to. [35] argues that an audience can relate better to the implications and complexity of something they are familiar with. The use of metaphors is one of the most common teaching techniques, especially when it is used to help children comprehend complex concepts. The same principle of using metaphors has also been used in computing as a way of making users, especially the naive ones, relate to and comprehend complex concepts. One of the most famous metaphors used in a software application is the *turtle* in LOGO [36], where the illustration of painting by following the movement of turtle's tail in the dust was very successful, especially with children, and it helped them learn and use LOGO more effectively.

Applications' interfaces that utilise metaphors chosen/drawn from the users' work environment can make concepts of such an application conceptually simpler to all users and particularly the naive ones. [37] suggests that the use of a 'red book' as a metaphor can be successful in the interface design of an application that is developed for the employees of a company where a "real" red book is used for a specific purpose. Considering that a significant number of database applications are business applications and that the majority of these applications' users are familiar with office environment make us believe that the use of office metaphors can be advantageous when used in such database applications' interfaces. Thus, we agree with [38] that it is conceptually simpler for an office employee to work with a database interface that allows him/her to relate to pictures from his/her work (i.e. office) environment.

However, the use of metaphors requires careful consideration, as choosing the wrong metaphor can easily confuse users and lead them to misinterpret the intended semantics with possibly disastrous results. For example [39] describes a case where the action of ejecting a disk was represented as moving a disk icon into a wastebasket; the metaphor used was clearly wrong and confusing, as users interpret it as throwing garbage into a wastebasket. In conclusion, a wisely selected/used metaphor that a target audience can relate to, can conceptually simplify complex concepts.

3.3 Graphical Scheme Design for Different Database Models

A graphical scheme is a representation of structures of the underlying database model. Designers of such schemes maintain a one-to-one correspondence between elements of the graphical scheme and the corresponding constructs of the underlying database scheme. An immediate implication is that, a graphical

scheme representation is constrained by the constructs of the underlying database model and it could not represent the database scheme of another database model.

If the graphical scheme is to be independent of the underlying database model, the set of metaphors used must be independent of the structures of the underlying database model. For example, in User's View the metaphor *folder* can be interpreted as either a class object or as an entity. Moreover, the graphical scheme's set of metaphors must cover all different database models' concepts or, if possible, cover all database models' concepts of a superset database model. In case that metaphors cover a superset model, then they can cover all the superset's subset database models. The OODBM is a superset database model because its concepts are a superset of the concepts and constructs of the relational, the nested relational, the complex objects and the semantic database models. Thus, a graphical scheme which covers the power of an OODBM, can support all subsets of OODBM model. The User's View is designed for the OODBM, in such a way that it can represent constructs of all subset database models of the OODBM.

3.4 The User's View of GOQL

The representation of all possible details of an underlying scheme can be overwhelming for some users so it may be preferable for certain details to be hidden from certain classes of users. Thus, hiding the underlying scheme's perplexing details combined with the use of appropriate metaphors can simplify the graphical scheme and makes its use more effective for naive users. Herein, we present such a graphical scheme, which we call the User's View. In Figure 1, the User's View of the running example is represented. The OQL code for the running example is given in Appendix I.

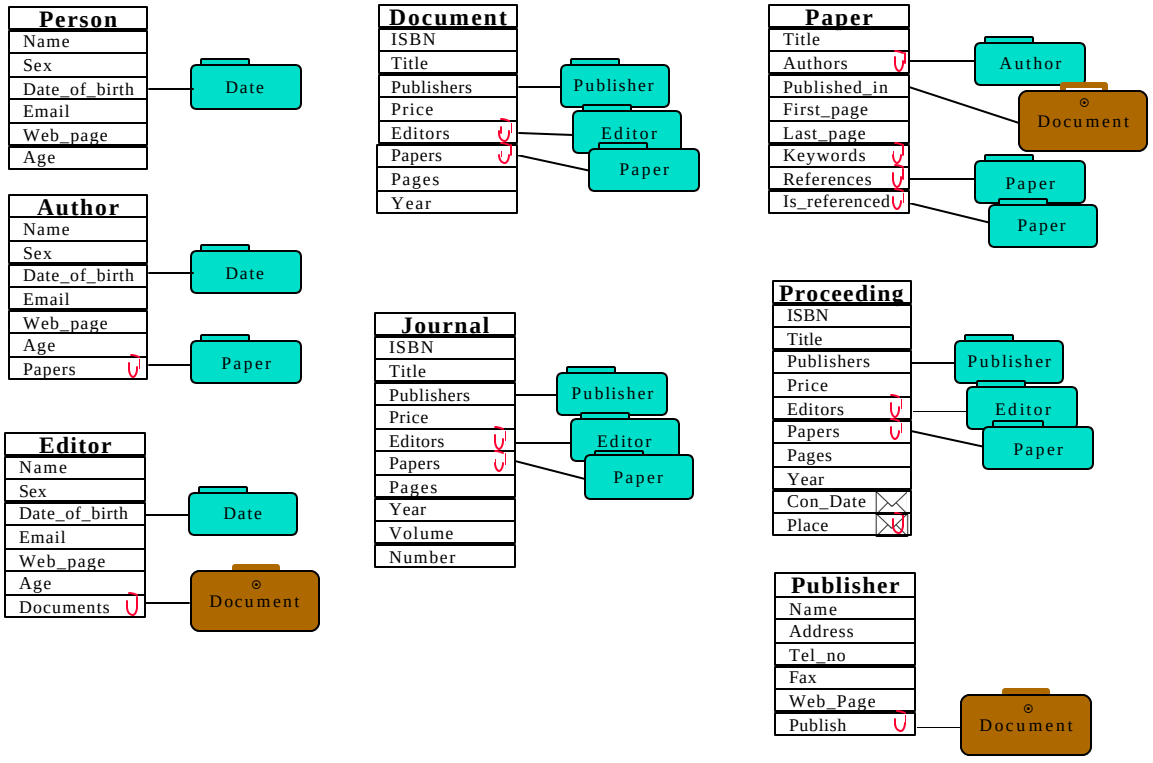


Figure 1 The User's View graphical representation of the running example database scheme

In particular, the User's View supports all the features of the database scheme of the underlying database model and metaphors are used to represent the features of this scheme. More specifically the constructs of the underlying database model are represented as follows:

- ***A Class/entity*** is represented as a table, with the name of the class/entity appearing in the header of the table.
- ***An Attribute*** of a class/entity is represented as a cell/row in the class/entity's table.

- ***A Method*** is also represented as an attribute; thus users are presented with one concept that covers both attributes and methods and which simplifies, the scheme representation without loss of any power.
- ***A Relationship*** is represented as a link between an attribute of a class/entity and a folder or briefcase. A folder is a metaphor that is used to represent all the objects of a particular class/entity that are related to the object at hand. The briefcase is a metaphor that is used to represent objects of more than one class/entity that are related to the object at hand; a briefcase is named by the name of the superclass of all these objects. An opened briefcase contains a number of 'folders' each of which represents the subclasses of the particular superclass. Users can either open a briefcase and select a subclass (folder) or they can select the whole superclass (briefcase) if it is required. A ***recursive*** relationship is a relationship that may exist between an attribute of a class/entity to the same class/entity, (for example in Figure 1 the property *Paper.References* is that of type Paper). A special appearance of a folder/briefcase is presented when the folder/briefcase is represented without a link to the corresponding class/entity attribute, i.e. the object is presented 'closed' without giving any details of the object's characteristics (like attributes, etc.).
- No particular symbol is used to represent a ***class hierarchy***, because class hierarchies are hidden in User's View and they are partially implied/used only when the briefcase metaphor is used. When a Users' View is created, every class/entity is represented as having all its properties and the properties that it inherits from all its superclasses. In this way, the User's View provides a much simplified representation of the underlying scheme without any loss of the OODBM power.
- An envelope placed inside the right edge of a property cell is used to represent a ***complex property***. Users can open such an envelope to reveal the hidden properties that comprise the complex property. The reason for choosing a closed envelope to represent complex properties is to avoid overloading the graphical scheme and to show that a property has something hidden in that envelope.
- ***A Collection property*** is represented by a paper clip placed inside the top right edge of a property cell. The aim of this metaphor is to show to a user that a property is a composite one, i.e. it is of set, bag, list or array type, without explicitly showing the type of the collection. The paper clip metaphor was chosen because it is used in offices to catch a wad of papers.
- Method parameters are represented by a disk metaphor, which when it is opened reveals the parameters of the method. The disk is placed inside the left edge of the method cell (see Figure 2). The disk metaphor was chosen because it is used in offices to give/store data (parameters) to computer applications.

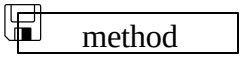


Figure 2: The method parameter metaphor

For the scheme design we used a set of desktop metaphors in order to simplify the picture of the scheme without loosing any of the database scheme power. Actually, the User's View is the only graphical scheme representation which represents graphically the method parameters. Sometimes a metaphor generalises the idea of the represented concept (e.g. briefcase, envelope, disk), in order to present a more effective scheme, giving at the same time to the user the possibility to open the metaphor and use the hidden objects.

3.5 GOQL and the Problem of Complex Databases – The Condensed View

GOQL deals with the problem of representing a complex database scheme by adopting a hybrid approach, which involves the top-down approach, the browsing approach, and the scheme simplification approach. Figure 3 contains a ***condensed*** graphical representation of the scheme of the running example database. It consists of a number of closed folders, one for each of the defined classes/entities. Users can choose to 'open' any of the included folders to examine the features of that class. By selecting a folder (Figure 4a) the graphical scheme is moving one level down (*top down* approach) for the particular class/entity (Figure 4b). At this level relationship *browsing* can allow a user to open any of the contained relationships, whereas using selecting elements of these relationships can allow a user to develop/navigate part of the scheme that is of his/her interest; in other words a *scheme simplification* is achieved by scheme developing (Figure 4). We believe that this combination of the three approaches provides a straightforward mechanism to browse schemes of even a complex database. Also, users can use this approach to construct subschemes of the initial

scheme that will meet the demands of a query. Examples of graphical queries based on the running example of this paper are given in Appendix II.

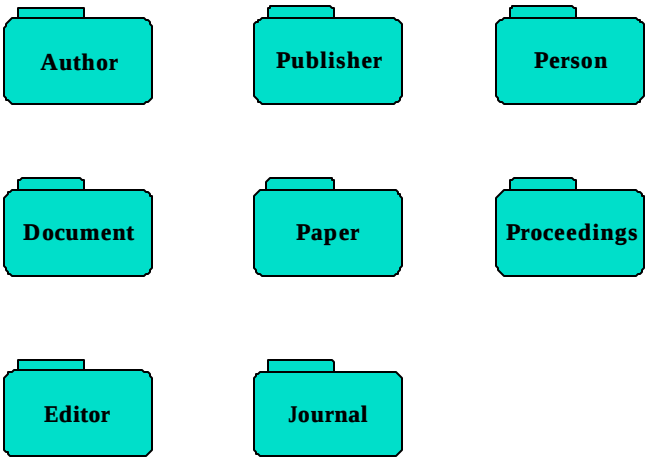


Figure 3: The condensed User's View

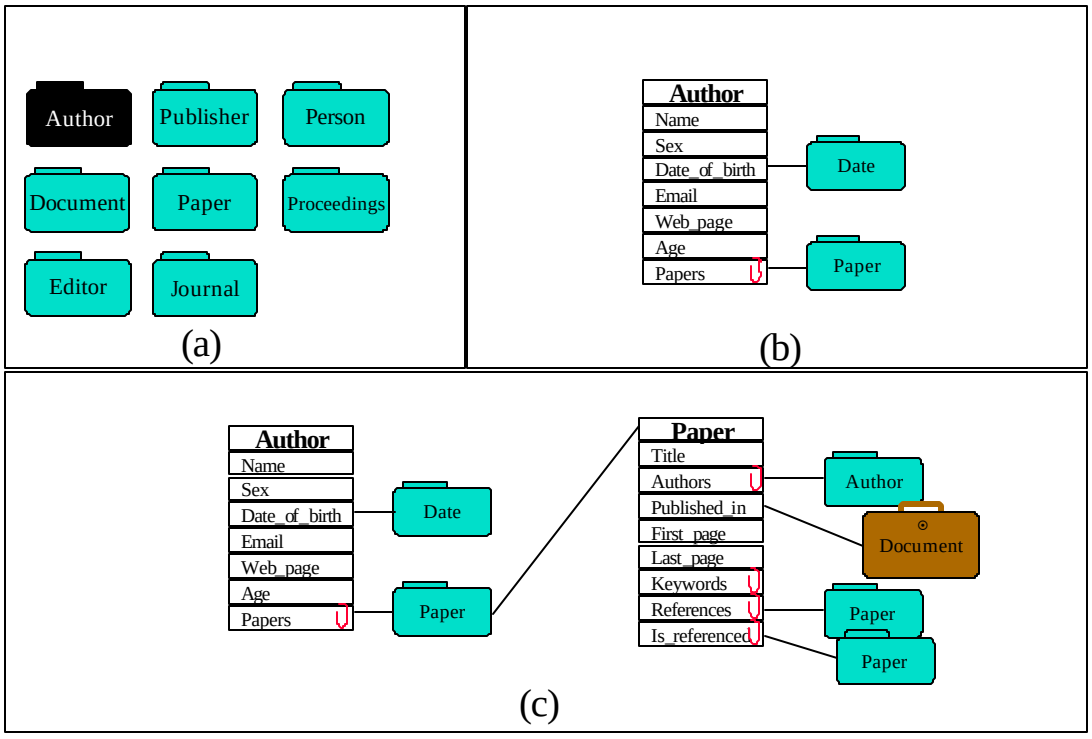


Figure 4: The Folders Window, Scheme Developing, and relationship browsing

4. GOQL Design Considerations

When designing the GOQL, we had as a target the implementation of a graphical query language, which would support the whole repertoire of the ODMG 3.0 OQL [15]. In addition, we wanted to provide expert users with a language that they could use more productively than OQL or other GQLs and naive users with a language that they would be able to use with the least possible training. To achieve these targets we used metaphors for the graphical representation of the scheme and we tried to give a visual look to the query

construction that will not trouble the user with perplexing symbols and diagrams but to highlight the important elements of the query icon using colour and some special symbols, which attract the attention of the user. Finally, every GQOL feature is a visual formalism.

4.1 Visual Representations

In [18] eleven (11) different types of visual representations were identified as being used in various graphical query languages. The three more important and commonly used visual representations are the form based one, the diagram based and the icon-based representation. A combination of two or more methods results in hybrid systems. So far according to [18], the combinations that have been adopted are the following: a) forms and diagrams, b) diagrams and icons and c) forms, diagrams and icons. Overall, hybrid systems produce better query representations because designers can choose the features of each representation that will create a better and more productive result. Furthermore, quite frequently users are used to specific pictures from their life, which are presented by different types of presentation; thus, it is very difficult to illustrate these pictures using only one technique. The above has led us to decide to use forms for the presentation of data, diagrams for the presentation of operators and icons for the representation of toolbars for the GOQL interface, resulting in a hybrid language.

4.2 Cognitive and Technical Aspects of the Design

GQLs are focused on specific classes of users and this is reflected by their design. GOQL is aimed at all three types of users, namely expert, skilled and naive; thus, the GOQL's design had to incorporate both cognitive and technical characteristics that address the needs of each of these classes of users. The cognitive target of the design is to create an easy to learn language with a natural language characteristic, which will be easily used. To achieve this, a number of desktop metaphors were used, especially for the representation of the graphical scheme. Also, for the representation of the various operators we employed simple shapes that have been selected having as criteria the type of the operator and the number of its operands. The overall aim was to allow users to intuitively recognise the semantics of the various language operators and use them accordingly. Finally, in order to enhance the 'readability' of the graphical queries, colour was used. For the technical aspects of the design we developed a language with a hybrid query constructing mechanism. To improve the query construction mechanism we incorporated in the design the majority of the known methods of formulating a query. Moreover, all of the GOQL's tools are *visual formalisms*, i.e. tools, that are formally defined to be used by a computer and which also can be visualised by users [40].

4.3 Shape

According to [41] the shape of elements of a language can be utilised as an effective way of differentiating between these elements. In GOQL, we tried to employ the above idea to differentiate between operators. Thus, we have classified operators to categories and we use different shapes to represent each such category. Operators within each category are identified by the category's symbol and word identifying the operator. The shapes we used are:

- Hexagons, which are used to represent boolean operators.
- Small ovals, which are used to represent unary operators.
- Large ovals, which are used to represent binary operators.
- Circles, which are used to represent sorting.

4.4 Colour

In devising a strategy for the use of colour in GOQL, we have followed the suggestions of [39, 42]. Even the monitor was adjusted to display only shades of grey to check whether the used colours, can be easily read by the majority of users. Thus, we designed all the GOQL features using only seven (7) colours. Moreover, we have chosen highly contrasting colours that are easily recognisable/readable in a monochrome monitor. Furthermore, we utilised a colour convention that we felt was intuitive as it is also used in a similar way in other areas such as a traffic/work environment [39, 42]. In particular, *red* was used for alerts, for example when function parameters have to be provided. *Green* was used to indicate that all is clear, i.e. no syntax error. *Yellow* was used as a sign of caution; the draw attention action is given by painted yellow *semicircles*, which indicate where a condition has been inserted. *Dark blue* was used to highlight selected items; in

particular we chose to use a dark colour to highlight projected items and the blue colour to achieve the differentiation for the projected items. We used different colours to present each of the metaphors that we introduced in chapter three. The colours used, which also give a nice result in a grey scale, are:

- For the folder, the turquoise green.
- For the briefcase, the brown.
- For the envelope, a mix of red and orange.
- For the clip, the light blue.

For the background used a shade of a grey, which is a neutral colour, i.e. it does not make colours painted on it to look darker or lighter; it is friendly and unobtrusive [41]. Finally, a thin black border is used with each of the defined tool-shapes of GOQL to make them clearly recognisable by all users.

4.5 Formulating the Query

The construction of a query has to be user-friendly without sacrificing any of the query language expressive power. In [18], four different methods of formulating a graphical query are discussed, in particular:

1 The *Scheme navigation*, allows users to navigate through a database scheme, by moving from one object/entity to another in order to find a required object/entity. There are three different approaches to scheme navigation, namely:

Arbitrary connected paths. This method is based on the graphical scheme of a database. A user navigates through the graphical scheme following an arbitrary path and produces a simplified scheme by selecting objects/entities along with their relationships and any other properties involved in the query. Any other components (projections, selections, …) of the query are defined using this simplified scheme. This method has been adopted in the Gql graphical query language [7].

Connected hierarchical paths. This method is based on the use of a basic concept of a database such as an entity or an object as the starting point in query construction. A GQL uses this basic concept as a root to create a hierarchical view of the underlying database. Users use the generated graphical representation to construct a query. The above technique is commonly used with GQLs for the object-oriented database systems because of the structure of the model. This method has been adopted in the graphical query language SUPER [11].

Unconnected paths. This approach is used mainly with relational databases, where a user can choose a number of tables from a database scheme. The tables chosen do not have to be related. To form a query the selected tables are linked by arbitrary value-based relationships, that may or may not exist in the stored database scheme. This method has been adopted in the graphical query language QBD* [2].

2 *Subqueries* are query expressions in their own right, but appear as parts of another query. The following are the main cases of subqueries:

i. *Composition of concepts*. Every part of a query can be considered as a query itself, i.e. it is comprised by a basic object, it has some projected properties, and it can be represented by an icon/diagram.

ii. *Use of stored queries*. These are queries or subqueries (views) the expression of which has been stored and which are represented either as icons/diagrams or expanded.

3 The *by matching* technique is used to check if any of stored data can be matched to a part of the answer is already known. Two are the main cases for that technique:

i. *By example*. In this case, in the query structure the user fills the known properties of a database and in the evaluation stage the query language looks for the values of the projected properties, which give a correct answer without violate any database rule.

ii. *Pattern matching*. This differ from the *by example* technique in that a user knows only a part of a value of the property and use a pattern to complete that value. (i.e. An example is the operator *like* of the SQL).

4 The *range selection* is another method for formulating queries. In this method users set a range of numeric or alphanumeric values that database objects must satisfy. We believe that the range

selection is more appropriately utilises a GQL interface if users are allowed to choose the required boundaries from a list of values.

To formulate queries in GOQL, we use a combination of the above techniques. In particular,

- The relationship property introduces scheme navigation; a relationship type property linked at the right side by an arbitrary edge to another object.
- Hierarchical paths in a graphical scheme are hidden but users are still given the impression that a query is constructed using a hierarchical view. Query construction start with an object (the *basic object*), which is the root of a query and all the other features of a query are linked as leaves on that root.
- GOQL supports subqueries; this is done with the use of the *frame* tool, and the use of stored queries.
- The query selection is defined by using both of the 'by matching' techniques, and the 'range selection' used in the interface.

4.6 Visual Formalisms

In [40] a graphical representation has to be based on visual formalisms. These have the following two basic characteristics: (a) they are visual because they are generated, comprehended and communicated by humans and (b) formal because they are manipulated, maintained and analysed by computers. The GOQL design philosophy is based on visual formalisms.

5. The GOQL Interface

The User Interface of GOQL (UIGOQL) was developed based on the concepts discussed in the previous sections. The first step, before any query is constructed in GOQL, is to load the underlying database metadata. This involves the use of a simple dialog box. Following the loading of the relevant metadata, users have the option to access the graphical scheme representation of the underlying database either in a detailed form (Users View Window) or in a condensed form (Folders Window). These two forms are used throughout the query construction, either in consultive way for the database structure or for copying an object of a specific class on the Query Window (QW) canvas. The QW is created either by opening a stored query or by starting a new query, by double clicking on an object appearing in the Folders Window (FW). The query construction takes place in the QW using the available tools. Finally, the graphical query is translated into o2 OQL [43].UIGOQL is available for all types of users. Thus, UIGOQL has four basic characteristics, namely:

1. Metaphors, to make the interface more natural to the user.
2. Colour, to emphasise features of the query.
3. Help, in various depths.
4. Error mechanism, employed to propose a solution for the query construction errors.

5.1 The Database Scheme

When loading GOQL, users are presented with the main GOQL window (Figure 5). The first step for constructing a GOQL query is to load the underlying database metadata. This involves a dialog process during which users choose from a list of available databases the one they require to use. Following their choice, users are presented with a folders window and the User's View window.

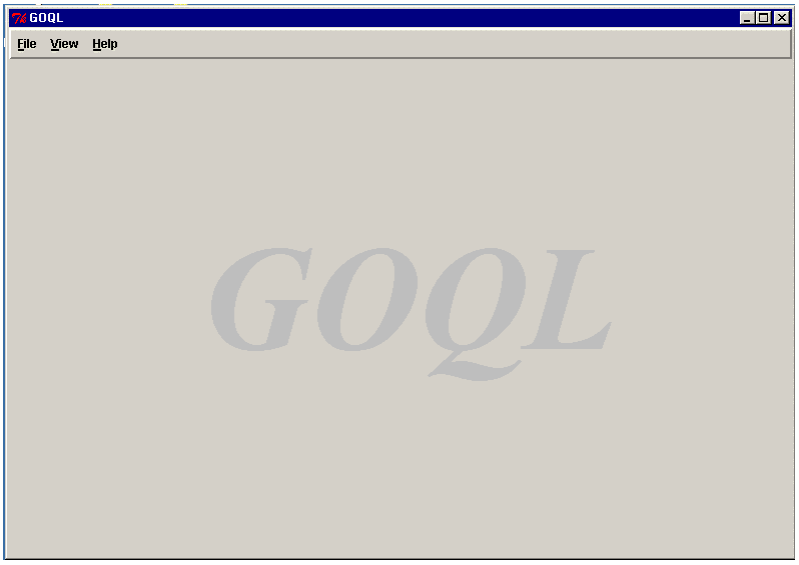


Figure 5: The GOQL main window

5.2 Loading the Metadata

To load the metadata users are presented with a dialog box (Figure 6), which is activated by selecting the *Open* option from the *File* pop-menu of the GOQL main window (Figure 5). The dialog box consists of three components, namely: the label, which displays at the top of the dialog box the path to the current directory; the 'UP' button, which users can use to move a level up in the directory structure; and the basic environment, which is comprised by two windows. The left window displays the names of directories defined within the current directory, by double clicking the left mouse on a directory name users can descend a level in the directory structure and move to the chosen directory. The right window displays all the o2 database files with the suffix '.load'. A user loads a database by double-clicking on the database name with the left button of the mouse. Finally, a user can close a database and any open query window by using the *close* menu option of the *File* pop-up menu of the GOQL main window (Figure 5).

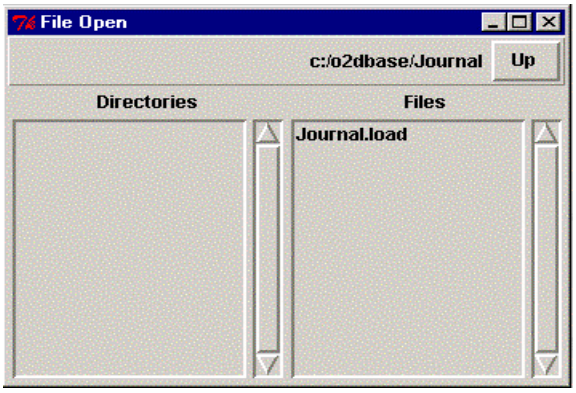


Figure 6: The Load Database Dialog Box.

Following the selection of the required database users are presented with the 'Folders Window'. Users can also choose to display the 'Users' View' Window.

5.3 The Folders Window

The Folders Window contains all the objects of the chosen database. Each such object is represented by a closed folder icon (Figure 7). The FW is created either by opening a database or by selecting the *Folders* option from the *View* pop-up menu of the GOQL main window. The main purpose of the FW is that it serves as the starting point for a query construction. Users can 'open' a folder by double clicking on it. Each 'opened' folder from the FW can become the root object for a subquery. Finally, whenever a user clicks the right mouse button on an object icon the name of that object class is displayed.

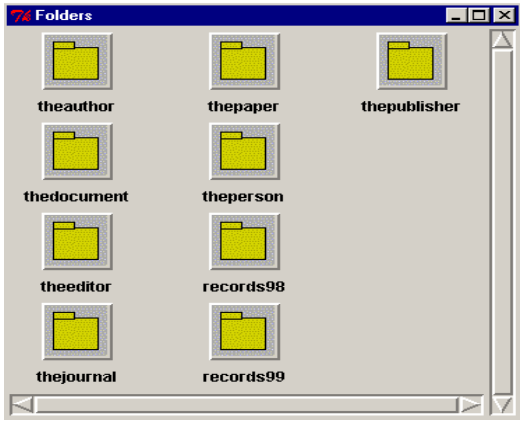


Figure 7: The Folders Window

5.4 The Users' View Window

The Users' View Window (UVW) is a detailed graphical representation of the underlying database scheme. The UVW is created by selecting the *User's View* menu option from the *View* pop-up menu of the GOQL main window. Users can consult the UVW at any time during the query construction process. Figure 8 contains part of the Users' View of the running example.

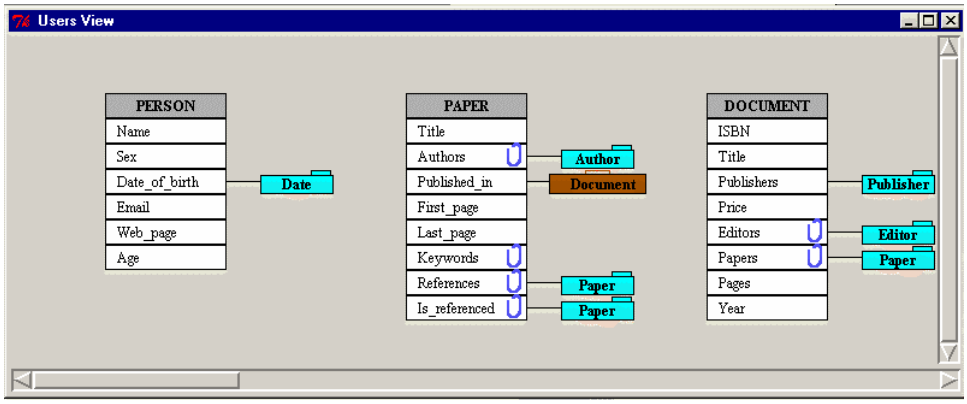


Figure 8: The Users' View window

5.5 The Query Window

The *Query Window* is the window where users can construct query expressions. Users will be presented with a QW if they select a root object from the FW or if they choose to open an already stored query expression. When the QW opens, users are presented with a horizontal toolbar, a vertical toolbar, a menu bar, a canvas and a message line (Figure 9).

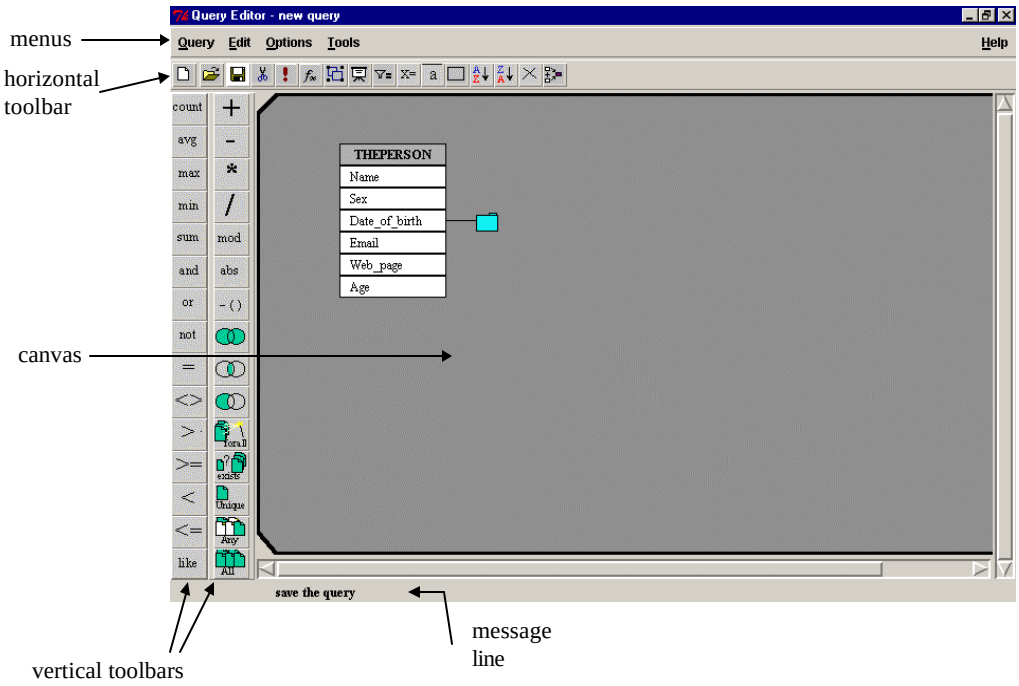


Figure 9: The Query Window

The canvas is the area where the query construction takes place and it is where the root object that was selected from the FW is displayed on. The message line is used by GOQL to display explanatory messages. All the constructs/tools/functions that GOQL offers for the construction and manipulation of queries are organised and made available through the menu bar and the two toolbars. The menu bar contains five pull-down menus, namely the Query menu, the Edit menu, the Options menu, the Tools menu and the Help menu. Selecting an option from any of these menus activates the action associated with the chosen option. Examples of GOQL queries are given in Appendix II.

5.6 The Toolbars and Menus

GOQL has two toolbars, both of them containing a set of buttons. Each button has a unique icon and an action/tool associated with it. Buttons correspond to a menu options and offer to users a faster way of invoking particular actions/tools than that of the pull-down menu. The icon of each button is a metaphor for the action/tool associated with the button. The metaphor used with each button has either been purposely designed or been selected because it has been commonly used in other well-established graphical user interfaces to represent the particular action/tool associated with this button. Selecting or 'pressing' a particular button involves placing the mouse pointer over it and clicking the left mouse button. Selecting a button activates the associated action/tool and makes the button appear on the user interface as being 'pressed'. Finally, whenever the focus of the mouse pointer is moved over a button the background colour of this button changes to white to highlight the event and an explanatory message about this button is displayed on the message line. The horizontal toolbar, Figure 10, contain sixteen buttons. Four of these buttons, namely the cut button, the highlight button, the pick button and the missing button have been designed for repeated use. Thus, when they are selected they will remain selected until a different button is selected.

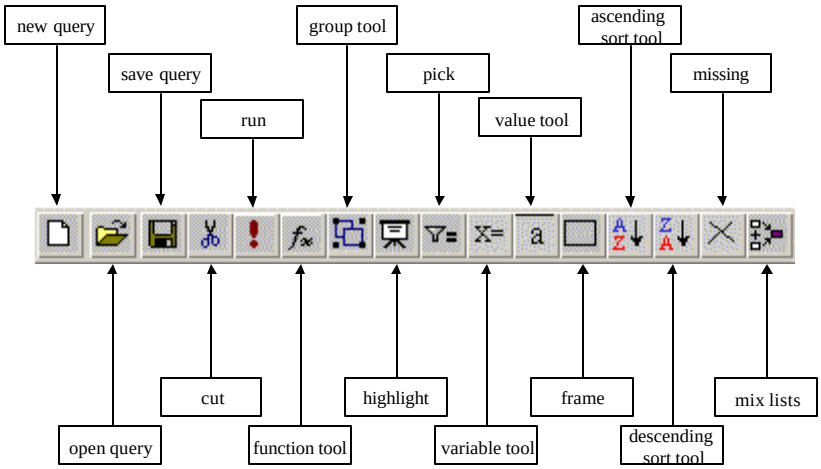


Figure 10: The horizontal toolbar in the Query Window

The vertical toolbar consists of two columns each of which contains fifteen buttons, Figure 10. These buttons of the vertical toolbar are organised according to their functionality into nine groups. In particular, the aggregate functions group comprising the 'count', the 'avg', the 'max', the 'min' and the 'sum' button, the boolean operators group comprising the 'and', the 'or', and the 'not' button, the comparison operators group comprising the '=', the '<>', the '>', the '>=', the '<', the '<=', and the 'like' button, the arithmetic operators group comprising the '+', the '-', the '*', the '/', and the 'mod', the absolute operator group comprising the 'abs' button, the negative operator group comprising the '-' button, the set operators group comprising the 'union', the 'intersect' and the 'except' button, the quantifying operators group comprising the 'for all', the 'exists' and the 'unique' button and the inclusive quantifying operators group comprising the 'any' and the 'all' button.

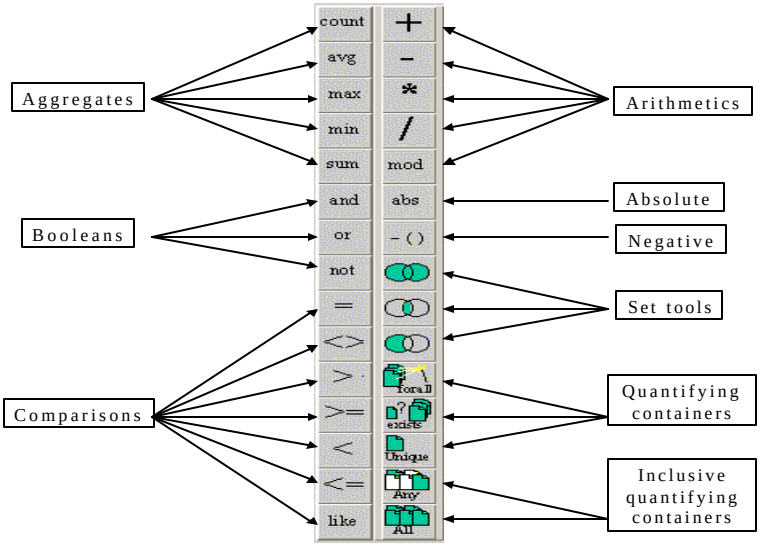


Figure 11: The vertical toolbar in the Query Window

6. GOQL's System Architecture and Implementation

GOQL runs on the o2 Object-Oriented Database Management System [43] and was implemented using the o2 system and Tcl/Tk [44, 45]. In particular, Tcl/Tk was used for the implementation of the GOQL interface and the development of the GOQL translator. Tcl is an open source programming language that is based on the C programming language, whereas Tk is an open source language that provides developers of graphical user

interfaces with a library of functions/tools that accelerate the development of graphical user interfaces. The open source nature of Tcl/Tk and the portability of Tcl/Tk code across platforms were the main reason that influenced our decision towards their use.The choice of the underlying OODBMS was determined by

(a) the support of the ODMG OQL and

(b) the availability of such DBMS.

The main reason for the ODMG OQL compliance requirement was the portability of the GOQL system. Thus, the o2 DBMS [43] was used as the underlying DBMS, whereas o2C [43] was used as means of passing the produced OQL query to the underlying OODBMS for processing and for handling the results returned. Although, GOQL was implemented based on the O2 DBMS, its design is such that it can be ported to another DBMS platform that complies with the ODMG 3.0 with minimal effort by making minimal changes to the way the data structure is generated from the metadata of the underlying OODBMS.

Figure 12, gives a pictorial description of the GOQL System Architecture. GOQL consists of the Data Structure Generator, the Scheme Viewer, the Query Editor, the Error Handling Mechanism, the Help Mechanism and the Translator.

Before any GOQL query is constructed and run, the relevant database should be loaded in GOQL. The metadata of the underlying database is provided from the underlying OODBMs to GOQL's Data Structure Generator, which constructs the database GOQL's Data Structure. The Data Structure consists of a number of files that contain information about the scheme of the underlying DBMS. These files have the same structure regardless of the underlying OODBMs.

Once the Data Structure is constructed, the Scheme Viewer is used to generate the User's View Window and the Folders Window. The UV and the FW provide the starting point for developing queries. In more details, a user can start constructing, loading, editing, deleting, running and storing graphical queries using the Query Editor (QE). The user can load folders from the FW in the QE and start opening them and developing the query (Figure 13). During the query development, the user can consult the UV in order to understand better the underlying database scheme. The new query can then be saved through the QE in the GOQL Data Structure (Figure 14). Saved queries can then be opened by the QE and be edited and saved again.

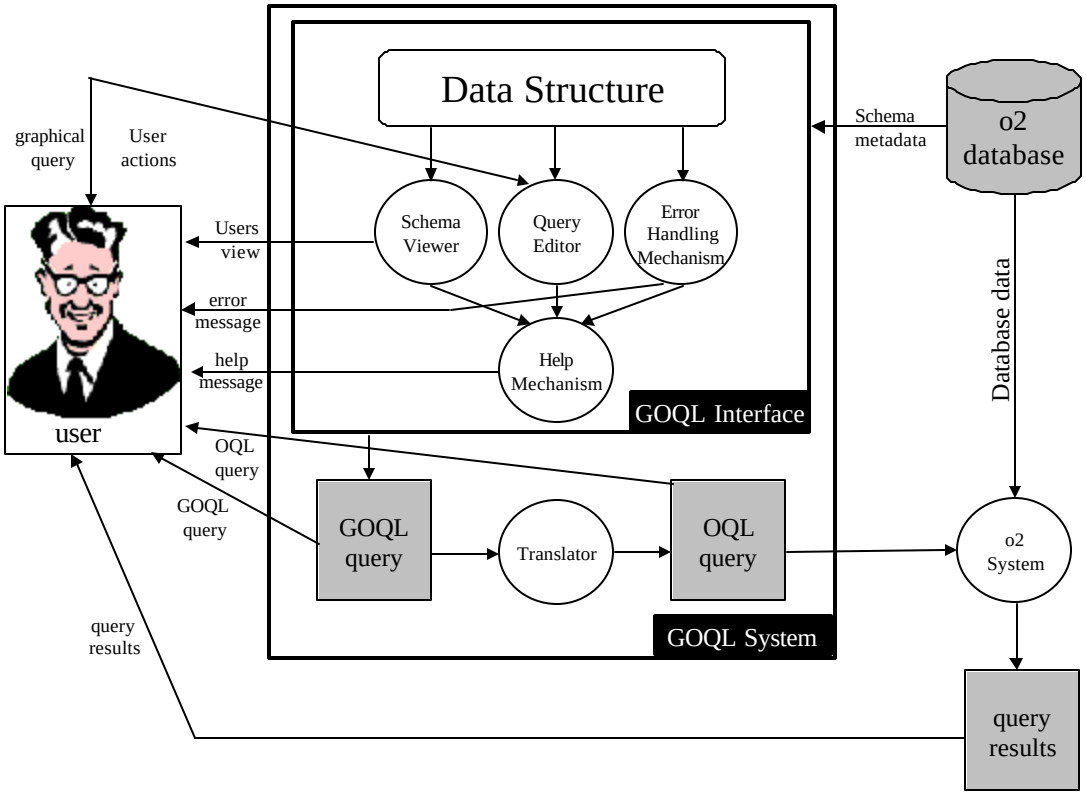


Figure 12: The Architecture of GOQL

During the query development, the Error Handling Mechanism (EHM) checks for errors and informs the user accordingly. More specifically, the EHM checks for syntactical errors (incomplete or invalid graphical queries), compatibility errors and opening file errors (e.g. attempting to open a query before the database metadata is loaded, attempting to open a query for another database scheme, etc.). The Help Mechanism (HM) provides further general and/or specific information about GOQL and features thereof and suggests solutions for errors that may appear. The HM is available through the UV, the FW and the QE.

Once a GOQL query has been developed and is syntactically correct, the user can run the query. This is done through the Translator that first transforms the GOQL query into an OQL query, which is displayed to the user (Figure 15). The user can then review the OQL query and hence revise the GOQL query if s/he thinks that there are some logical errors. The OQL query is then exported to the underlying OODBMs, which runs the query and produces the results.

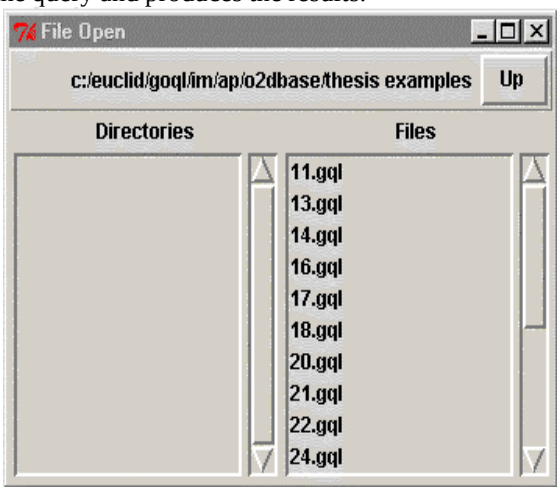


Figure 13: Opening a Query

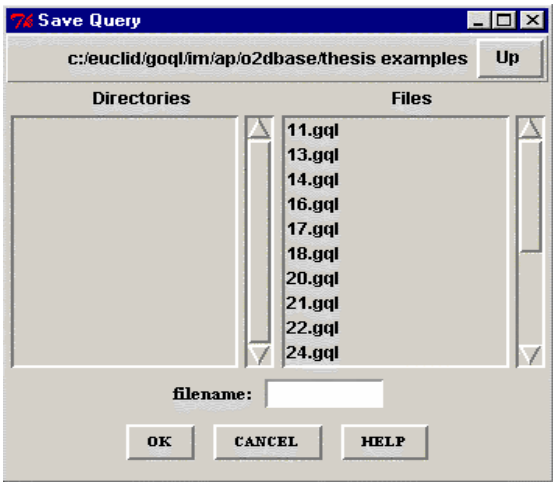


Figure 14: Saving a Query

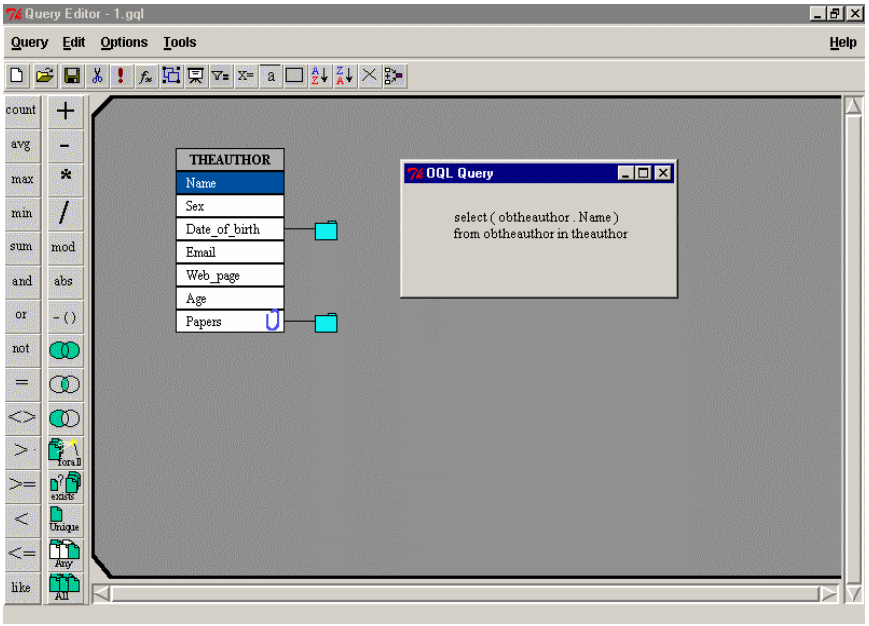


Figure 15: The Query Window

7. Formal Definition of the GOQL Object-Oriented Database Model

GOQL is based on the ODMG 3.0 [18] object-oriented database model. In this section, we provide a formal description of the GOQL object-oriented database model.

A GOQL *object-oriented database scheme S* is defined as a set of classes: S ::= {$Class_i$ | i = 0 ... n}

A Class C is defined as a triple: C ::= < State, Behaviour, Inheritance >, where

C.State ::= {Attributes, Relationships}
C.State.Attributes ::= {< $attribute_i$, $Type_i$> | i = 0…m, m is the number of attributes}
*C.State.*Relationships ::= {<$relationship_i$, $Object\_Type_i$> | i = 0…j, j is the number of relationships}
C.Behaviour ::= {$Methods_i$ | i = 0…n, n is the number of methods}
C.Behaviour.Methods ::= {<Parameter_list, Type>}
*C.Behaviour.*Methods.*Parameter_list* ::= {<$parameter_i$, $Type_i$>, | i = 0 ... j, j is the number of parameters in the method}
C.Inheritance ::= { C_i | i = 0 … k, k is the number of immediate superclasses of C }

In the ODMG 3.0 [15] model two types of inheritance are defined, namely EXTENDS and ISA. The EXTENDS inheritance defines the inheritance of the state from a superclass to a subclass; The ISA inheritance defines the inheritance of behaviour between a superclass and a subclass. In GOQL the inheritance is defined as the combination of the EXTENDS and the ISA inheritance; i.e. the inheritance of state and behaviour.

The underlying object-oriented database model of GOQL supports the types that are supported by the ODMG 3.0 database model. Herein, we provide the list of types supported:

Type ::= { Literal_Type
Object_Type

Literal_Type ::= { Atomic_Literal
Structured_Literal
Null_Literal

Atomic_Literal ::= { integer
float
boolean
character
String
Enumeration

Enumeration is a type generator, which defines a named literal type that can take on only the values lis ted in the declaration. It is defined as:

Enum_id = <eid_1, eid_2, …, eid_n>, eid_1, eid_2, …, $eid_n \in$ Literal_Type.

The Enum_id identifies the enumeration type and the eid_1, eid_2, …, eid_n, which are of the same Literal type are placed in an ordered sequence.

Structured_Literal ::= { Literal_ Structure
Literal_Collection

Literal_Structure ::= {< $attribute_i$, $Type_i$> | i = 0 .. m, m is the number of attributes in the stru cture}

Literal_Collection ::= { Set<t>
Bag<t>
List<t>
Array<t> , t ∈ Atomic_Literal

Each of these literal collections are type generators, which take as parameters the types shown in the brackets. In particular, a set type is denoted by the type generator *Set* and a type parameter *t*. Similar the other three literal collection types are denoted by the relevant type generator and a type parameter *t*.

A *Set* literal collection is an unordered collection of literal elements that do not allow duplicates. A *Bag* literal collection is an unordered collection of literal elements that allow duplicates. A *List* literal collection is a possible infinite sequence of elements that is either empty or has a head and a tail that is a list. An *Array* literal collection is a list but it is finite so it can be accessed by an ordinal number.

$$\text{Object\_Type} ::= \begin{cases} \text{Object} \\ \text{Structured\_Object} \end{cases}$$

Objects are instances of classes and are identified by object identifiers. An Object is defined by:

Object ::= {<Oid, C, V> | Oid is a unique object identifier, C is a class of the scheme S, the Object is an instance of C and V is the set of values for every attribute of the object's State, for every relationship of the object's Relationships and for every parameter of the object's method}.

$$\text{Structured\_Object} ::= \begin{cases} \text{Object\_Structure} \\ \text{Object\_Collection} \end{cases}$$

$$\text{Object\_Structure} ::= \{<\text{attribute}_i: \text{Object}_i> \mid i = 0..m, \text{ m is the number of attributes in the structure}\}$$

$$\text{Object\_Collection} ::= \begin{cases} \text{Set<t>} \\ \text{Bag<t>} \\ \text{List<t>} \\ \text{Array<t>} \end{cases} , t \in \text{Object}$$

Each of these object collections are type generators, which take as parameters the types shown in the brackets. They are similar to literal collection with the difference that the type parameter *t* is of type Object.

8. A Mapping from the Underlying OODBM Scheme to the UV

The UV is a graphical representation of a given ODMG 3.0 object-oriented database scheme. Moreover, the UV consists of a number of *UV_class_tables*, each of which represents a class of the given OODB scheme. Thus, to define the UV for a given OODB scheme, a mapping has to be defined that will detail how constructs of an OODB map on to these of the UV. Herein, we give a formal definition of this mapping. Firstly, we give some definitions:

Definition 8.1: The all_attributes operator returns the set of all attributes that are either defined explicitly in a class or inherited from any of its superclasses. Thus, if C is a class then,

$$\text{all\_attributes}(C) ::= \{A \mid A \in C.\text{State.Attributes}\} \cup \{\text{all\_attributes}(X) \mid X \in C.\text{Inheritance}\}$$

Definition 8.2: The all_relationships operator returns the set of all relationships that are either defined explicitly in a class or inherited from any of its superclasses. Thus, if C is a class then,

$$\text{all\_relationships}(C) ::= \{ R \mid R \in C.\text{State.Relationships}\} \cup \{ \text{all\_relationships}(X) \mid X \in C.\text{Inheritance } \}$$

Definition 8.3: The all_methods operator returns the set of all methods that are either defined explicitly in a class or inherited from any of its superclasses. Thus, if C is a class then,

```
all_methods(C) ::= {M | M ∈ C.Behaviour} ∪ { all_methods(X) | X ∈
C.Inheritance }
```

Definition 8.4: The all_properties operator returns the set of all attributes, relationships and methods that are either defined explicitly in a class or inherited from any of its superclasses. Thus, if C is a class then,

```
all_properties(C) ::= all_attributes(C) ∪ all_methods(C) ∪
all_relationships(C)
```

Definition 8.5: The all_superclasses operator returns the set of all superclasses that are either defined explicitly in a class or inherited from any of its superclasses. Thus, if C is a class then,

```
all_superclasses(C) ::= {M | M ∈ C.Inheritance} ∪ { all_superclasses(X)
| X ∈ C.Inheritance }
```

A GOQL object-oriented database scheme, *S*, is mapped onto a User's View scheme, *uvS*, by creating a *UV_class_table* for each class *C* in *S*. Each row (uvrow) of a *UV_class_table* corresponds to a property of the corresponding class in S.

Definition 8.6: The *uvrow* function returns a row of a *UV_class_table* which corresponds to a property of the corresponding class in S. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ⇒ uvrow(p)
```

Each row of a *UV_class_table* corresponds to a Type.

Definition 8.7: The *uvtype* function returns the type of a UV_class_table row. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type ⇒ uvtype(p)) = t
```

Definition 8.8: The *uvobject* function returns the object name of a UV_class_table row which is of type Object. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ uvtype(p)==Object ∧ (∃o ∈
Object) ⇒ uvobject(p) = o
```

Definition 8.9: The *uvclass* function returns the class name of a UV_class_table row which is of type Object. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ uvtype(p)==Object ∧ (∃o ∈
Object) ∧ uvobject(p)==o ∧ (∃Cl ∈ S) ⇒ uvclass(p)=o.Cl
```

A uvrow represents a row of a UV_class_table without its type. The type of a uvrow is defined by the use of five functions.

Definition 8.10: A *folder* function defines a uvrow that is of type either Object or Object_Structure. If the uvrow is of type Object then the class of uvrow does not have any superclass. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ (((uvtype(p) == Object) ∧
(∃p') (p' == uvclass(p)) ∧ all_superclasses(p') == nil)∨ (uvtype(p) ==
Object_Structure)) ⇒ folder(p)
```

Definition 8.11: A *briefcase* function defines a uvrow that is of type Object. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ ((uvtype(p) == Object) ∧
(∃p') (p' == uvclass(p)) ∧ (all_superclasses(p') ≠ nil)) ⇒ briefcase(p)
```

Definition 8.12: A *clip row* defines a uvrow that is of type either Literal_Collection or Object_Collection. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ ((uvtype(p) ==
Literal_Collection) ∨ (uvtype(p)==Object_Collection)) ⇒ clip(p)
```

Definition 8.13: An *envelope row* defines a uvrow that is of type Structure_Literal. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ ((uvtype(p) ==
Structure_Literal) ⇒ envelope(p)
```

Definition 8.14: A *simple row* defines a uvrow that is of type either Atomic_Literal or Null_Literal. Thus, if p is a property of a class C then,

```
(∀ p) (p ∈ all_properties(C)) ∧ (∃t ∈ Type) ∧ ((uvtype(p) ==
Atomic_Literal) ∨ (uvtype(p) == Null_Literal)) ⇒ single(p)
```

Definition 8.15: A *disk row* represents a method that takes some parameters. If p is a property of a class C then,

```
(∀ p) (p ∈ all_methods(C)) (∃pm ∈ parameter_list(p)) ⇒ disk(p,pm)
```

A formal definition of the mapping is given below:

```
∀C ∈ S, ∃ uvC ∈ uvS
Let p ∈ all_properties(uvC)
uvrow(p)
If (p ∈ all_method(C) ∧ ∃ pm ∈ p.parameter_list) then
      disk(p,pm)
endif
If (uvtype(p) == Object_Collection ∨ uvtype(p) == Literal_Collection)
then
      clip(p)
endif
If (uvtype(p) == Atomic_Literal ∨ uvtype(p) == Null_Literal) then
      single(p)
endif
If (uvtype(p) == Literal_Structure) then
      envelope(p)
elseif ((uvtype(p) == Object_Structure) ∨ ((uvtype(p) == Object) ∧
       ((∃p') (p'== uvclass(p)) ∧ (all_superclasses(p') == nil)))
      folder(p)
elseif ((uvtype(p) == Object) ∧ ((∃p') (p'== uvclass(p)) ∧
             (all_superclasses(p') ≠ nil)))
      briefcase(p)
endif
```

9. Conclusion

This paper presented the GOQL language. Amongst the most important advantages of GOQL is the database model independence that it supports and its simplified user interface that hides any perplexing details of the underlying model(s). The language's user interface, namely the User View and Folders Window, was explained and examples of graphical queries were given to illustrate the use and expressive power of the language. The paper presented a survey of existing graphical query languages and compared GOQL with the presented languages. It also addressed the design ideas taken into consideration during the interface design and briefly explained the architecture of the system. Finally, the paper presented formal specifications of the GOQL language which allow the translation of an ODBMG model into GOQL and the definition of formal rules for the syntax analysis of the language and the transformation of GOQL queries into OQL and vice versa. GOQL is fully functioning and is running on top of the o2 DBMS. Our current work involves continuous evaluation and maintenance of the system, involving correction of bugs and further enhancements of the system. We are also currently researching into CAD applications and interfaces so as to integrate some of their ideas/features into our language.

References

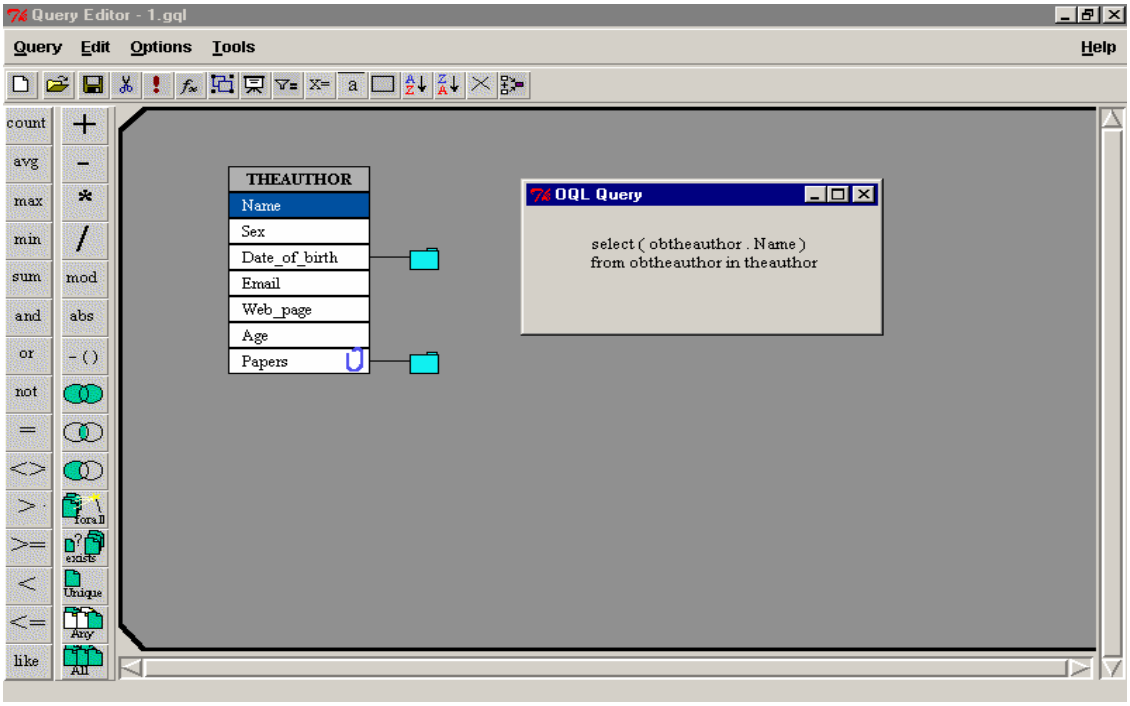
[1] P. P. Chen, The entity-relationship Model: toward a unified view of data, Journal of ACM Transactions on Database Systems, 1(1) (1976) 166 – 192.
[2] M. Angelaccio, T. Catarci, G. Santucci, QBD*: A Graphical Query Language with Recursion, IEEE Transaction on Software Engineering, 16(10) (1990) 1150-1163.
[3] Y.S. Jun, S.I. Yoo, , GOMI: A Graphical User Interface for Object-Oriented Databases, International Conference on Object-Oriented Interface Systems (OOIS), (1995) 238-251.
[4] J. Boyle, S. Leishman, M.D. Gray, From WIMPS to 3D: The Development of AMAZE, Journal of Visual Languages and Computing, 7(3) (1996) 291-319.
[5] M. Chavda, P. T. Wood, Co mbining Constraints and Data-Flow in A Visual Query Language, IEEE Symposium on Visual Languages, Capri (Italy), (1997) 125-126.
[6] N. Murray, N. Paton, C. Goble C., Kaleidoquery: A Visual Query Language for Object Databases, the 4th IFIP Working Conference on Visual Database Systems - VDB 4, L'Aquila, Italy, (1998) 247-257.
[7] A. Papantonakis, Gql, a Declarative Graphical Query Language Based on the Functional Data Model, PhD Thesis, Birkbeck College, University of London, 1995.
[8] J. Paredaens, P. Peelman, L. Tanca, G-Log: a graph-based query language, IEEE Transactions on Knowledge and Data Engineering, 7(3) (1995) 436-453.
[9] H. Kim, H.F. Korth, A. Silverschatz, PICASSO: A Graphical Query Language, Software-Practice and Experience, 18 (3) (1988) 169-203.
[10] M. Kuntz, R. Melchert, Pasta-3's Graphical Query Language: Direct Manipulation, Co-operative Queries, Full Expressive Power, the 15th International Conference on Very Large Databases, Amsterdam, (1989) 97-105.
[11] Y. Dennebouy, M. Andersson, A. Auddino, Y. Dupont, E. Fontana, M. Gentile, S. Spaccapietra, SUPER: Visual Interfaces for Object + Relationship Data Models, Visual Languages and Computing, 6(1) (1995) 73-99.
[12] S. Dar, N.H. Gehani, H.V. Jagadish, J. Srinivasan, Queries in an Object-Oriented Graphical Interface, Visual Languages and Computing, 6(1) (1995) 27-52.
[13] J. Bowen, Formal Specification & Documentation Using Z: A Case Study Approach. International Thomson Computer Press, 1996.
[14] P. Wegner, The Vienna definition language, ACM Computing Survey, 4(1) (1972) pp. 5-63.
[15] R.G.G. Cattell, D.K. Barry, The Object Database Standard: ODMG 3.0, Morgan Kaufmann Publishers, 2000.
[16] E. Georgiadou, E. Keramopoulos, Measuring the Understandability of a Graphical Query Language through a Controlled Experiment, of the BCS International Conference of Software Quality Management, Loughborough, April 18-20, (2001) 295-307.
[17] N.H. McDonald, J.P. McNally, Query Language Feature Analysis by Usability, Computer Languages, 7(3/4), (1982) 103-124.
[18] T. Catarci, M.F. Costabile, S. Levialdi, C. Batini, Visual Query Systems for Databases: A Survey, Visual Languages and Computing, 8 (2) (1997) 215-260.

[19] E. Keramopoulos, P. Pouyioutas, T. Ptohos, Comparison Analysis of Graphical Models of Object-Oriented Databases and the GOQL Model, the 6th WSEAS International Conference on Computers, Rethymno, Crete Island, Greece, (2002).
[20] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modelling Language User Guide, Addison-Wesley Object Technology Series, 1998.
[21] Rational Software Whitepaper, UML for Data Modeling Profile. http://www.rational.com/products/whitepapers/index_all.jsp, 2002.
[22] P. Coad, E. Yourdon, Object-Oriented Analysis, Prentice Hall, 2nd Edition, 1991.
[23] G. Schiffer, Scheuermann, Multiple Views and Abstractions with an Extended-Entity Relationship Model, Computer Languages, 4, (1979) 139-154.
[24] S. Khoshafian, Object-Oriented Databases, John Wiley & Sons Inc, 1995.
[25] N. Murray, C. Goble, N. Paton, A Framework for Describing Visual Interfaces to Databases, Visual Languages and Computing, 9 (4) (1998) 429-456.
[26] M. Angelaccio, T. Catarci, G. Santucci, Query by Diagram*: A Fully Visual Query System, Visual Languages and Computing, 1(2) (1990) 255-273.
[27] A. Papantonakis, P.J.H. King, Syntax and Semantics of Gql, a Graphical Query Language, Visual Languages and Computing, 6(1) (1995) 3-25.
[28] M. Chavda, P. T. Wood, Towards an ODMG-Compliant Visual Object Query Language, the 23rd VLDB Conference, Athens (Greece), (1997) 456-465.
[29] Microsoft Corporation Ltd, Microsoft Office 2000 Resource Kit, Microsoft Press, 1999.
[30] SOFT TOOLRACK LTD, GQL (Graphical Query Language), 1995.
[31] Business Objects Ltd, Available at:http://www.businessobjects.com/products/query_report_analysis.htm, 2003
[32] Oracle White Paper, Oracle Designer: Technical Overview. Available at: http://otn.oracle.com/products/designer/pdf/9idesigner_overview.pdf, 2002
[33] ISO, Database Language – SQL (ISO 9075:1992(E)). International Organization for Standardization, 1992.
[34] A. Massari, S. Pavani, L. Saladini, QBI: An Iconic Query System for Inexpert Users, the Advanced Visual Interfaces Workshop (AVI'94), Italy, (1994) 240-242.
[35] O. Torgny, Metaphor – a working concept, the Contextual Design- Design in context, Stockholm, April (1997) 3-14.
[36] Slack James, 1990. *Turbo Pascal With Turtle Graphics*. West Publishing Company.
[37] Lovgren J., 1994. How to choose good metaphors. In *Journal of IEEE Software*, Vol. 11, No 3, pp. 86-88.
[38] Collins D., 1995. *Designing Object-Oriented User Interfaces.* Benjamin/ Cummings Publishing Company Inc.
[39] Dix, J. Finlay, G. Abowd, R. Beale, Human Computer Interaction (Second Edition), Prentice Hall Europe, 1998.
[40] D. Harel, On visual formalism, the Communication of the ACM , 31(5) (1988) 514-530.
[41] J. Foley, A. van Dam, Computer Graphics Principles and Practice, Second edition, Addison Wesley Publishing Company, 1990.
[42] W.M. Newman, M.G. Lamming, Interactive System Design, Addison-Wesley Publishing Company, 1995.
[43] o2 Technology, o2 User Manuals, 1995.
[44] J.K. Ousterhout, Tcl and the Tk Toolkit, Addison-Wesley Professional Computing Press, 1995.
[45] E.F. Johnson, Graphical Applications with Tcl & Tk, M&T Books, 1996.

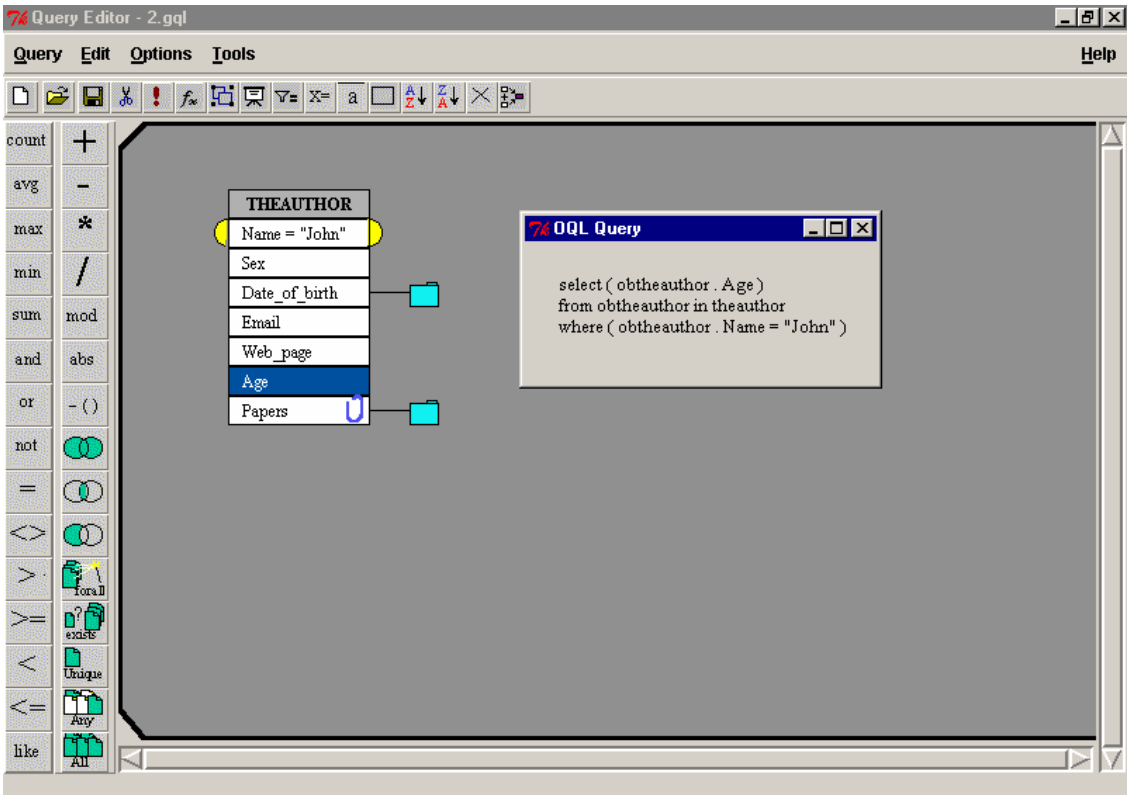
APPENDIX I – The ODMG Database Scheme of the Running Example of the Paper

```
class Document
(extent Thedocuments)
{
        attribute string ISBN;
        attribute string Title;
        relationship Publisher Publishers
            inverse Publisher::Has_published;
        attribute float Price;
        relationship set <Editor> Editors
            inverse Editor::Documents;
        relationship set <Paper> Papers
            inverse Paper::Published_in;
        attribute integer Pages;
        attribute integer Year;
};
class Journal extends Document
(extent Thejournals)
{
        attribute integer Volume;
        attribute integer Number;
};
class Proceedings extends Document
(extent Theproceedings)
{
        attribute struct Con_Date {Date Start_date, Date
End_date};
        attribute struct Place {set <string> City, string
Country};
};
class Publisher
(extent Thepublishers)
{
        attribute string Name;
        attribute string Address;
        attribute string Tel_no;
        attribute string Fax;
        attribute string Web_page;
        relationship set<Document> Publish
            inverse Document::Publishers;
};
class Person
(extent Thepersons)
{
        attribute string Name;
        attribute string Sex;
        attribute Date Date_of_birth;
        attribute string Email;
        attribute string Web_page;
        integer age;
};
class Editor extends Person
(extent Theeditors)
{
        relationship list<Document> Documents
            inverse Documents::Editors;
};
class Author extends Person
{
        relationship list<Paper> Papers
            inverse Paper::Authors;
};
class Paper
(extent Thepapers)
{
        attribute string Title;
        relationship set <Author> Authors
            inverse Author::Papers;
        relationship Document Published_in
            inverse Document::Papers;
        attribute integer First_page;
        attribute integer Last_page;
        attribute set<string> Keywords;
        relationship set<Paper> References
            inverse Paper::Is_referenced;
        relationship set<Paper> Is_referenced
            inverse Paper::References;
};
```

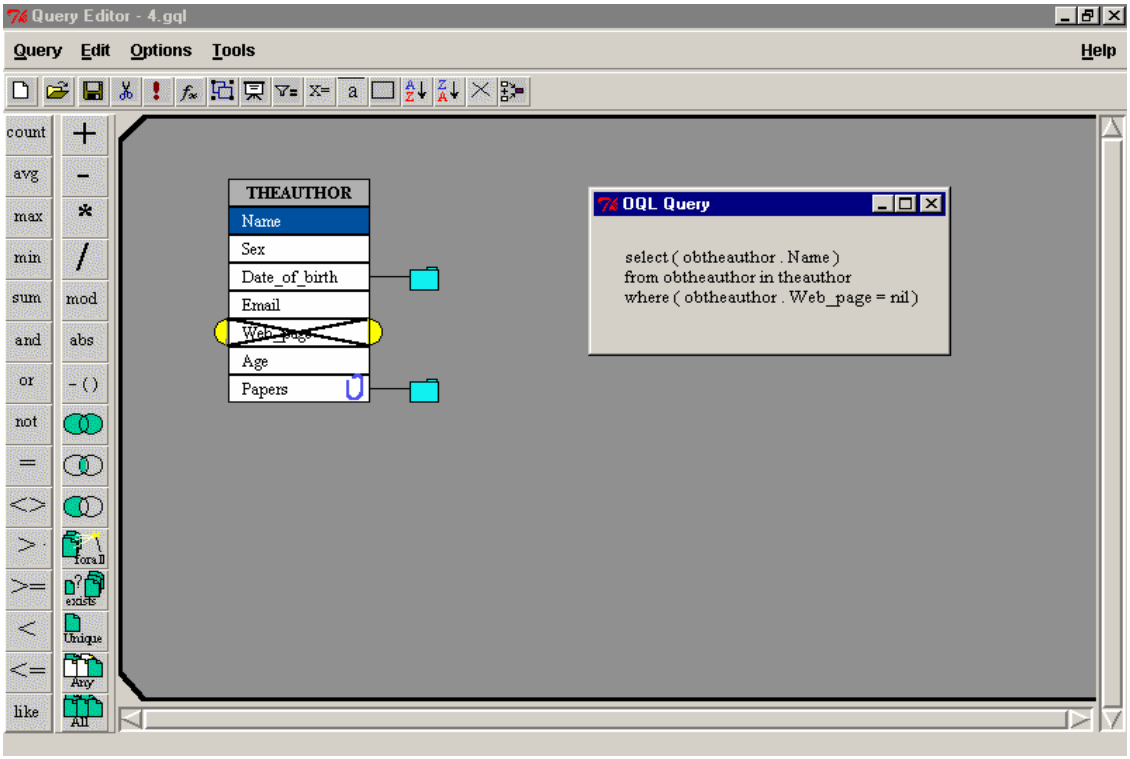
APPENDIX II – Examples of GOQL Queries



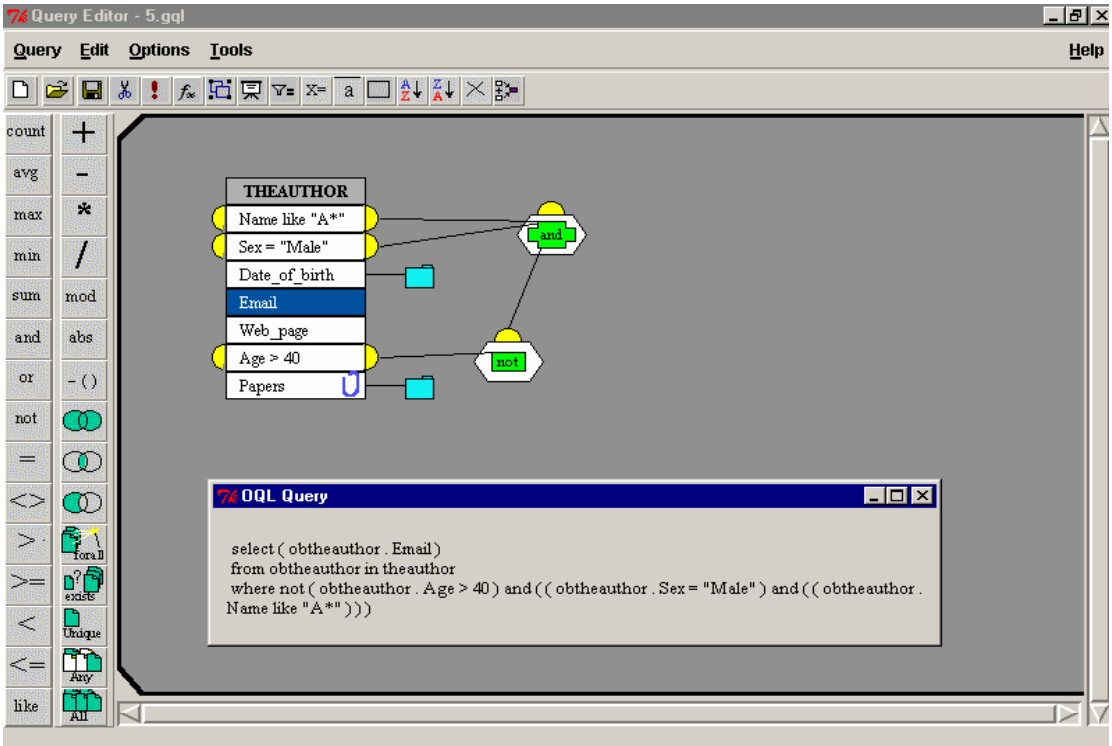
Query 1 Display the names of all the Authors.



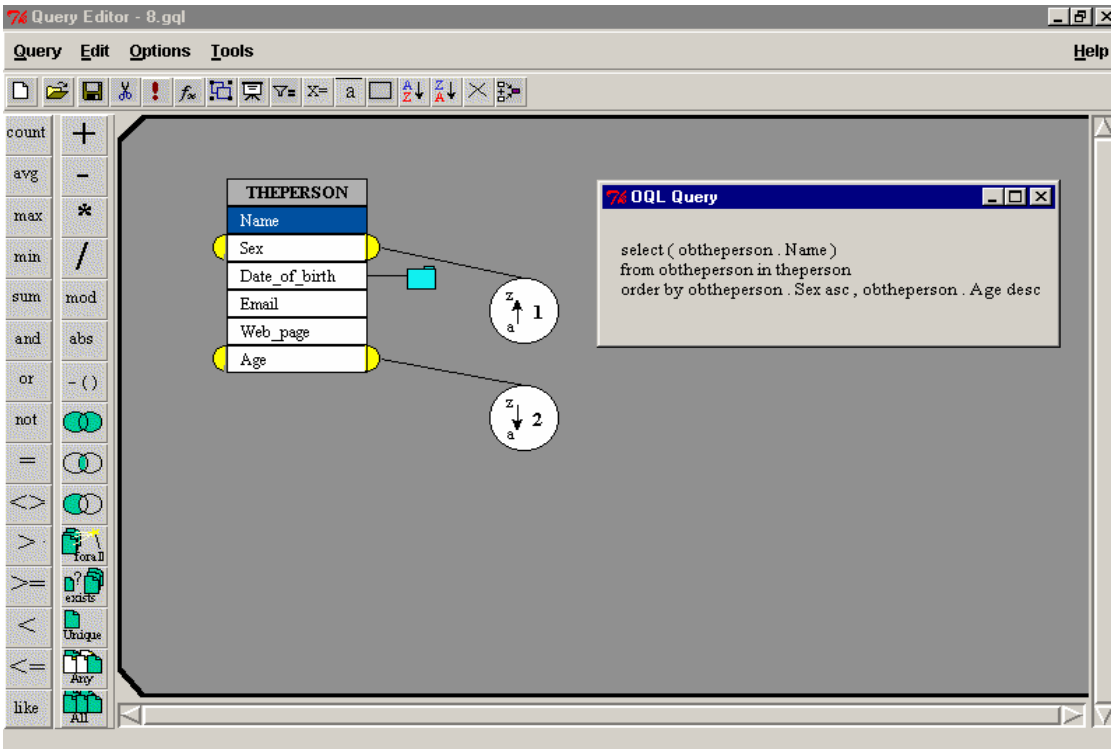
Query 2 Display the age of those Authors whose name is ‘John’.



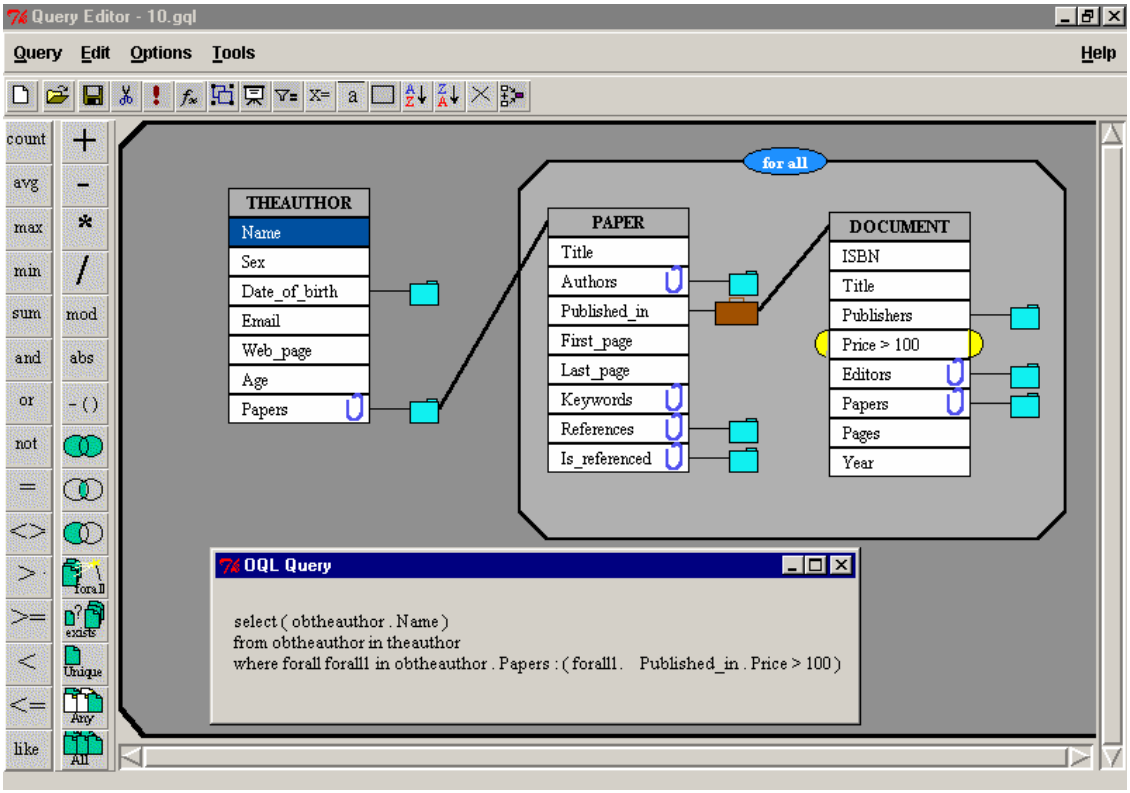
Query 3 Display the names of Authors who do not have a web_page.



Query 4 Display the email address of male Authors whose age is not greater than 40 and whose name stars with ‘A’.



Query 5 Display the names of persons in ascending order of their gender and descending order of their age.



Query 6 Display the names of authors whose published papers only in documents priced more than £100.