A formal definition of the Users View (UV) of the Graphical Object Query Language (GOQL).

Euclid Keramopoulos[1]
Philippos Pouyioutas[2]
Tasos Ptohos[1]

[1]Cavendish School of Computer Science
[2]School of Computing, Intercollege, Nicosia, Cyprus

A Formal Definition of the Users View (UV) of the Graphical Object Query Language (GOQL)

E. Keramopoulos [*], P. Pouyioutas [**], T. Ptohos [*]
euclid@it.teithe.gr pouyioutas.p@intercollege.ac.cy tasos@wmin.ac.uk

[*] Cavendish School of Computer Science, University of Westminster, 115 New Cavendish Str, London W1W 6UW, UK
[**] School of Computing, Intercollege, 46 Makedonitissas Ave., P.O. Box 24005, Nicosia 1700, CYPRUS

ABSTRACT

In this paper we provide a brief formal definition of the Users View (UV) of the Graphical Object Query Language (GOQL). The UV provides a graphical representation for object-oriented database schemas and hides from end-users most of the perplexing details of the object-oriented database model, such as methods, hierarchies and relationships. In particular, the UV does not distinguish between methods, attributes and relationships, encapsulates the is-a hierarchy and utilises a number of desktop metaphors to present a graphical schema that is easy to be understood by end-users. Thus, the UV provides the environment, through which end-users, can pose ad-hoc queries through GOQL.

We first give a brief formal definition of an object-oriented database schema in the GOQL model. This is given, by providing a formal definition of the basic element of such a schema, namely the class. The UV is then briefly formally defined as a mapping from a GOQL object-oriented database schema. Using this mapping, any object-oriented database schema can be translated into a graphical representation in the UV. The running example of the paper is used to demonstrate the mapping from the textual schema to the graphical schema of the UV. The formal definition of the UV will allow us, in the future, to formally define the graphical constructs of GOQL.

Keywords: Graphical Query Languages, Object-Oriented Databases, Formal Definition

1. Introduction

The evolution of database query languages during the last thirty years is strongly related to the evolution of database models and database systems. In the early days, i.e. the file processing systems era, the manipulation of data stored in such systems depended on programs written in some third generation programming languages. Similar imperative languages were also used for querying the hierarchical and network database systems that followed the file processing systems. The introduction of relational database systems proved a major leap forward in the design/use of query languages as it led to the development and use of declarative query languages.

The development of *declarative* query languages, such as QUEL and SQL, meant that users were able to query a system by describing *what* they wanted rather than *how* to get what they wanted. The declarative nature of these languages made them easier, for the end-user, to use and understand and, thus, it allowed them to be established as "user friendly" query languages suitable for both expert and naive users. Tabular and graphical query languages, which were based on the QBE paradigm [1], were also proposed for relational database systems. In languages such as Dbase, Paradox and Access, the required tables are displayed on the user's screen; users can usually express their queries by check marking projected attributes, inserting selection criteria in the appropriate attributes, and performing any join operations by inserting common example variables in the join attributes, or drawing lines between the join attributes.

Recent developments in database systems have resulted in the appearance of object-oriented databases, which also support the use of declarative query languages. The SQL-like object-oriented query language of the ODMG 2.0 model [2], namely OQL, has been widely accepted as the standard in the field.

Recent advances in the use of graphical interfaces and the successful introduction of the Graphical User Interfaces (GUIs) by a number of vendors, such as Apple and Microsoft, have also led to the development of a number of graphical programming languages, such as Visual C++ and Visual Basic. The popularity of graphical programming languages might be attributed to users' perception that these languages are easier to use and learn compared to textual languages.

The phenomenal success of graphical programming languages combined with researchers' drive to provide database users with an interface that will allow them to pose ad-hoc queries, led to the development of a number of graphical interfaces for various database systems. A comparative analysis of the features supported by such interfaces can be found in [3]. Typically, such interfaces allow queries to be visualised and be represented diagrammatically or graphically rather than in some obscure code.

In this paper, in Section 2, we present the graphical interface of the GOQL language, namely the Users View (UV), based on which GOQL queries can be posed [3,4,5,6]. An example of an object-oriented database schema and its corresponding UV is given to illustrate the graphical interface provided by GOQL. In Section 3, we provide a formal definition of the GOQL object-oriented database model. In Section 4, 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. The Users View (UV)

The language GOQL [3,4,5,6] has been designed to address the needs of end-users. It provides a graphical interface, namely the *Users View (UV)*, which hides and encapsulates some features of the underlying database and represents some others using metaphors. The language is based on the object model of the ODMG 2.0 and provides a graphical querying mechanism. Because there is a direct correspondence between the features of GOQL and OQL (GOQL supports all the features of OQL), the language can be used as an alternative graphical interface to OQL. Thus, GOQL allows users to express graphically queries ranging from simplistic ones to rather complicated ones. Among the features provided/supported by the language are: 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.

The graphical representation provided by GOQL is called *Users View (UV)*, and allows the representation of all the features of the underlying ODMG object model; however, a UV hides from users most of perplexing details, such as methods, hierarchies and relationships. In particular, a UV (a) does not distinguish between methods, relationships and attributes; (b) does not support an explicit representation of the *is-a* hierarchy lattice; instead properties inherited by a subclass are explicitly represented in that subclass as properties of the corresponding class table; (c) utilises a number of desktop metaphors that allow the representation of the other features of the object model.

The UV of a GOQL object-oriented database schema is generated from the stored metadata of the underlying schema, and it is comprised of a number of UV_class_tables (one UV_class_table for each class of the schema). Each UV_class_table contains a list of all the attributes, relationships and methods of the corresponding object-oriented database schema class, including the attributes, relationships and methods of all of the superclasses of the class under consideration. More specifically, the UV is a mapping of a GOQL object-oriented database schema into the graphical interface of GOQL, achieved by the use of various metaphors [3] as follows:

- each class is mapped into one *UV_class_table*
- each attribute of a class is mapped into a *property (row)* of the corresponding UV_class_table
- each method of a class is also mapped into a *property (row)* of the corresponding UV_class_table
- each relationship is also mapped into a *property (row)*; thus, there is no distinction, as the user is concerned, between an attribute, a method and a relationship; the resulting row is linked with a *folder* or a *briefcase* (a briefcase is used when the related class has subclasses)
- each attribute/method of any superclass of the mapped class is mapped into a *property (row)* of the corresponding UV_class_table; thus, inheritance is hidden from the user and the user sees all properties of any superclasses as properties of the UV_class_table
- types are hidden from the user (thus properties do not show their types), except the ones listed below, which help users to understand better the database schema:
    - a structure type is shown by an *envelope* which is placed in the right edge inside the row of the property; a user can open the envelope to reveal the properties that constitute the structure type
    - a collection type (set, bag, list or array) is shown by a *paper clip* which is placed in the top right edge inside the property row.

We next give in Figure 1 and Figure 2 a GOQL object-oriented database schema (expressed in OQL) and its corresponding Users View.

```
class Document
(extent Thedocuments)
{       attribute string ISBN;
        attribute string Title;
        relationship Publisher Publishers
       inverse Publisher::Publish;
        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
(extent Theauthors)
{      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;
};
```

Figure 1: A GOQL Object-Oriented Database Schema

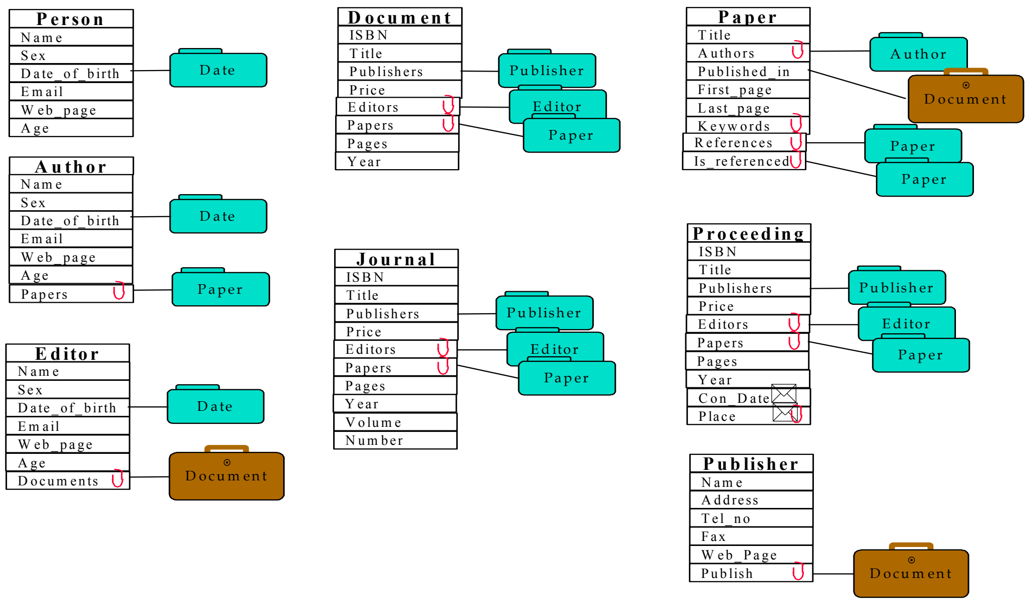

Figure 2: The Users View of the Schema

3. A formal definition of the GOQL object-oriented database model

GOQL is based on the ODMG 2.0 [2] object-oriented model. In this section, we provide a brief formal description of the GOQL object-oriented database model.

A GOQL *object-oriented database schema S*, is defined as set of classes:

S = {Class$_i$ | i = 0 .. n}

A Class C, is defined as a quadruplet:

C = <State, Behaviour, Relationships, Inheritance >, where

C.State = {< attribute$_i$: type$_i$> | i = 0 … m, m is the number of attributes }

C.Behaviour = { method$_i$ | i = 0 .. n, n is the number of methods }

C.Relationships = {relationship$_i$ | i = 0 .. j, j is the number of relationships }

C.Inheritance = { C$_i$ | i = 0 .. k, k is the number of immediate superclasses of C }

GOQL supports the types supported by on the ODMG 2.0 database model. We then provide the list of types supported:

Type =	{	Literal_Type
		Object_Type

Objects are identified by their object identifiers whereas Literals do not have identifiers.

Literal_Type =	{	Atomic_Literal
		Structured_Literal
Atomic_Literal =	{	integer
		float
		boolean
		character
		String
		Enumeration
Structured_Literal =	{	Literal_Collection
		Literal_Structure
Literal_Collection =	{	Set Type
		Bag Type
		List Type
		Array Type
Literal_Structure =		< attribute$_i$: Type> | i = 0 … m, m is the number of attributes in the structure >

Enumeration is a type generator, which defines a named literal type that can take on only the values listed in the declaration. *Sets* are unordered collections that do not allow duplicates. *Bags* are unordered collections that allow duplicates. *Lists* are ordered collections that allow duplicates. *Arrays* are unordered lists that can be located by position.

Object_Type =	{	Atomic_Object (instance of a user-defined or a built-in class)
		Structured_Object
Structured_Object =	{	Object_Collection
		Object_Structure
Object_Collection =	{	Set Type
		Bag Type
		List Type
		Array Type
Object_Structure =		< attribute$_i$: Type> | i = 0 … m, m is the number of attributes in the structure >

Atomic_object is either an instance of a user-defined or a built-in class (Date, Time, Timestamp and Interval).

4. A mapping from the GOQL object-oriented database schema to the GOQL Users View (UV)

As explained in Section 2, the Users View (UV) of a GOQL object-oriented database schema is generated from the stored metadata of the underlying schema, and it is comprised of a number of UV_class_tables (one UV_class_table for each class of the schema). Herein we give a formal description of this mapping. We first give some definitions, which are used in the mapping.

Let C be a class. The all_attributes operator returns the set of all attributes of the class C, which are either explicitly defined in the class or in any of its superclasses.

$$\text{all\_attributes (C)} = \{ \text{A} \mid \text{A} \in \text{C.State or (A} \in \text{all\_attributes (X) and X} \in \text{C.Inheritance)} \}$$

Let C be a class. The all_methods operator returns the set of all methods of the class C, which are either explicitly defined in the class or in any of its superclasses.

$$\text{all\_methods (C)} = \{ \text{M} \mid \text{M} \in \text{C.Behaviour or (M} \in \text{all\_methods (X) and X} \in \text{C.Inheritance)} \}$$

Let C be a class. The all_relationships operator returns the set of all relationships of the class C, which are either explicitly defined in the class or in any of its superclasses.

$$\text{all\_relationships (C)} = \{ \text{R} \mid \text{R} \in \text{C.Relationships or (R} \in \text{all\_relationships (X) and X} \in \text{C.Inheritance)} \}$$

A GOQL object-oriented database schema *S* is mapped into a Users View schema *uvS* by creating a *UV_class_table* for each class *C* in *S*. The *UV_class_table* has one *row* for each attribute/method/relationship of *C* (directly defined or inherited by any superclass of *C*). Five types of rows exist: *simple*, *envelope*, *clip*, *folder* and *briefcase* depending of the type of the attribute/method/relationship that resulted in the said row. An *envelope row* is one resulted from a *structured* attribute/method, a *clip row* is one resulted from a *collection* attribute/method/relationship, a *folder row* is one resulted from a relationship with a class which has no subclasses and finally a *briefcase row* is one resulted from a relationship with a class which has subclasses. More formally, this mapping can be defined as follows:

```
∀C ∈ S, ∃ uvC ∈ uvS: all_properties(uvC) = all_attributes(C) U all_methods(C) U all_relationships(C)
        the all_properties operator gives all the rows of a UV_class_table
Let p ∈ all_properties(uvC), p’ be the attribute/method/relationship resulted in p and t the type of p’

If ( t = Atomic_Object  and not ∃ C’ ∈ S: t ∈ all_superclasses(C’) ) then
                folder_row(p)
elseif ( t = Atomic_Object  and ∃ C’ ∈ S: t ∈ all_superclasses(C’) ) then
        briefcace_row(p)
elseif ( t = Object_Structure or t =  Literal_Structure ) then
        envelope_row(p)
elseif ( t = Object_Collection or t = Literal_Collection ) then
        clip_row(p)
        If ( unnest(t) = Atomic_Object  and not ∃ C’ ∈ S: unnest(t) ∈ all_superclasses(C’) ) then
                folder_row(p)
        elseif ( unnest(t) = Atomic_Object  and ∃ C’ ∈ S: unnest(t) ∈ all_superclasses(C’) ) then
                briefcace_row(p)
        elseif ( unnest(t) = Object_Structure or unnest(t) = Literal_Structure ) then
                envelope_row(p)
        endif
endif

where unnest(t) = collection_unnest(t) and (t = Object_Collection or t = Literal_Collection)
                = t, otherwise
        collection_unnest(t) = collection_unnest (t’), if (t’ = Object_Collection or t’ = Literal_Collection)
                             = t’, otherwise
                             where t = Object_Collection t’ or t = Literal_Collection t’
```

5. Conclusions

In this paper we provided a brief formal definition of an object-oriented database schema in GOQL. We have also explained the transformation of a given schema in the Users View (UV) graphical representation of GOQL. The graphical schema provides end-users with an interface that allows them to easily pose ad-hoc queries

[3, 4, 5, 6]. The transformation of the schema to its Users View has also been formally defined as a mapping from the schema to the UV. Our work is based on the ODMG 2.0 model [2]. The recent release of the ODMG 3.0 model [7] necessitates that we extend our work to be compatible with the new standard. Thus, our current work concentrates on amending the work presented herein, as well as amending the GOQL query language [3, 4, 5, 6] to be consistent with the ODMG 3.0 model. Furthermore, our current work concentrates on the methods and the behaviour part of the GOQL model. An evaluation experiment [8] carried out on the current version of GOQL provided us with optimism as regards with the usefulness and acceptability of the language.

References

[1] Zloof, M.M.: Query By Example. In *Proceedings of NCC, 44, AFIPS Press*, 1977.

[2] Cattell R.G.G. & Barry D.K. (Eds.). *The Object Database Standard: O.D.M.G. 2.0.* Morgan Kaufmann Publishers, 1997.

[3] Keramopoulos, E., Pouyioutas, P., & Ptohos, T. A Comparison Analysis of Graphical Models of Object-Oriented Databases and the User's View Level of GOQL. *To appear*, *International Conference on Information Visualisation* 2001.

[4] Keramopoulos, E., Pouyioutas P. & Ptohos T. The GOQL Graphical Query Language. *To appear, International Journal of Computers and Applications,* 2001.

[5] Keramopoulos, E. Ptohos, T. & Pouyioutas P., 2000. GOQL - A Graphical Query Language for Object-Oriented Databases. *IASTED AI2000 International Conference (Applied Informatics),* pp.129-133, Innsbruck, February 2000.

[6] Keramopoulos, E., Pouyioutas, P. & Ptohos, T. The User's View Level of the GOQL Graphical Query Language. *International Conference on Information Visualisation 99,* pp. 81-86, London, July 1999.

[7] Cattell R.G.G. & Barry D.K. (Eds.). *The Object Database Standard: O.D.M.G. 3.0.* Morgan Kaufmann Publishers, 2000.

[8] Georgiadou E., Keramopoulos E. Measuring the Understandability of a Graphical Query Language through a Controlled Experiment. *BCS Conference of Software Quality Management, Loughborough*, 18-20 April 2001.