Fundamental Graphical Primitives for Visual Query Languages[1]

(Accepted for pubblication on Information Systems Vol. 18 No. 2)

Tiziana Catarci♣, *Giuseppe Santucci*♣, *Michele Angelaccio*♦

♣Dipartimento di Informatica e Sistemistica
Universita' degli Studi di Roma "La Sapienza"
Via Salaria, 113 - 00198 Roma, Italy

♦Dipartimento di Ingegneria Elettronica
Universita' degli Studi di Roma II "Tor Vergata"
via O. Raimondo 38 - 00173 Roma, Italy

ABSTRACT

The need of a friendly man-machine interaction is becoming crucial for a large variety of applications. In order to reach such a friendliness a new class of languages has been proposed (Visual Languages), based on the extensive use of graphical and iconic mechanisms. We are interested in a particular subclass of Visual Languages, called Visual Query Languages (VQLs), devoted to the extraction of information from databases. VQLs are mainly based on the idea of applying new interaction mechanisms, based on the "direct manipulation" paradigm, on visually represented databases. Various VQLs have been proposed, but only a few of them are provided with a formal definition and, also when such a formal definition exists, it does not give the semantics of the graphical operations performed by the user. In this paper we aim to provide such a semantics by proposing a graphical data model, the graph model, in which the visual representation is part of the model itself, and a minimal set of Graphical Primitives, in terms of which general query operations may be visually expressed. Moreover, we show that: a) such a model may be used as a general visual representation for the most common data models; b) the Graphical Primitives have the same expressive power as well-known query languages; c) the graph model and the Graphical Primitives may be used as basic constituents of more complex existing visual representations and visual query languages, thus giving them a semantics independent from the underlying data model.

[1] Research partially supported by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, National Research Council, Italy and by EEC under the Esprit Project 6398 VENUS.

1. Introduction

The effective use of different complex application programs is strongly hindered by the needed knowledge of operational modes, the user technical background, and the information on both the system application domain and its interaction mechanisms. Currently, a casual user of a system is generally discouraged from working with it, since the amount of technical knowledge required for running a program generally needs a long learning time and/or an on-line tutor. Recently, a number of interfaces based on different techniques that better exploit human senses have been suggested and implemented so enlarging the bandwidth of the man-machine communication channel. As an example, attempts have been made to create interfaces based on the use of natural language, both in form of textual dialogue and of speech interpretation and synthesis. Moreover, the availability of graphical devices at low cost has given rise in the last years to a large diffusion of interfaces using visual techniques. The well-known advantages of using images instead of text turn out to be particularly relevant in man-machine interaction. As a further advantage, the visual approach attracts user attention and stimulates a full coverage of all the available system facilities.

Recently, the database area has proved to be particularly fruitful for applying visual techniques specifically in accessing stored data. One reason is that very often the database is queried by a casual user, who is not necessarily acquainted with languages such as SQL [Date 1987]. Moreover, with respect to using a restricted natural language, both the dependence upon the native language of the user and the limitations imposed by the application area are avoided. Finally, the typical process of query formulation itself encourages using visual techniques. In fact, as illustrated in [Batini et al. 1991], such a process can be seen as constituted by three phases: in the first phase the user selects the part of the database s/he wants operate on; in the second phase s/he defines the relations within the selected part in order to produce the query result; in the third phase s/he operates on the query result. The first phase is the best candidate for a visual representation, since it can be seen as a kind of zoom on the database in order to select the parts of interest. In the second phase the main operations are used to express conditions on the structural relations according to the formulated query. A visual representation here is useful in order to show both the database contents and the query structure. Finally, in the third phase, it may be useful to visually represent the structured relations for analyzing the query result.

Visual Query Systems (*VQSs*) may be defined as query systems essentially based on the use of visual representations to depict the domain of interest and express the related requests. VQSs provide user-friendly query interfaces for accessing a database. They include both a language to express the queries in a pictorial form (i.e., a *visual query language*, *VQL*) and a variety of functionalities to facilitate man-machine interaction. The VQSs are oriented to a wide spectrum of users who have limited technical skills and generally ignore the inner structure of the accessed database. In the last years, many VQSs have been proposed in the literature, adopting a range of different visual representations and interaction strategies. However, the main part of any VQS is constituted by the VQL it is based on.

In order to more precisely define a VQL, we refer to Chang's classification [Chang 1989], in which the *visual programming languages* are introduced. Visual programming languages handle objects that do not necessarily have a visual representation (e.g., traditional data types like arrays, lists, stacks and application oriented data types like forms and documents). In order to improve and facilitate man-machine interaction, the objects are presented visually; therefore, a visual representation is associated with these abstract objects. The programming constructs and the rules for combining them are also represented visually . Within the class of visual programming languages, a subclass exists which deals with a particular kind of non visual objects, i.e., data in databases; this subclass of languages is the one of visual query languages. Based on which visual element they mainly adopt, we have graphical and iconic VQLs, i.e., languages based on the extensive use of diagrams and icons respectively. Typically, iconic languages have a higher metaphorical power with respect to graphical ones. However, no standard can yet be imposed on the set of icons used in different applications. Moreover, graphical languages are more suited to be formalized, given the precise mathematical structure (i.e., the graph) the diagrams are based on.

Such a formalization allows for comparing them with traditional query languages, and precisely evaluating their expressive power.

Various graphical VQSs have been proposed (a survey is in [Batini et al. 1991]), but only a few of them are provided with a formal definition (see [Cruz Mendelson Wood 1988; Nanni 1988; Angelaccio Catarci Santucci 1990b; Consens and Mendelzon 1990]). All these systems are mainly based on the idea of proposing new visual representations for the classical, non-visual database models, together with new interaction mechanisms founded on the "direct manipulation" paradigm [Shneiderman 1983]. For example, we can consider the algebraic definition of relation in the relational model and represent it by using either a hypergraph or a table; in the same way we can represent the Relational Algebra operators [Codd 1972] by some navigation in a diagram. Unfortunately, this kind of approaches has produced a lack of formalization in defining the visual interaction mechanisms.

In order to overcome such a drawback, a possible solution is to unify the data model and its graphical representation, directly applying on it the graphical operations (such as selection of nodes and drawing of edges) having their own semantics (a proposal is in [Gyssens Paredaens Van Gucht 1990]). The approach presented in this paper is in this direction. The basic idea is in defining a minimal set of well-founded *Graphical Primitives*, in terms of which more complex visual interaction mechanisms may be precisely defined. The Graphical Primitives are defined on the basis of a graphical data model, the *graph model*, in which the visual representation is part of the model itself. Moreover, we will show in the following that the graph model and the Graphical Primitives provide the general principles for devising graphical interfaces to heterogeneous databases, expressed in terms of the most common traditional data models. Finally, since both the graph model and the Graphical Primitives consist of elementary graphical elements, it is possible to use them as basic constituents of more complex existing visual representations (i.e., E-R diagrams, object networks, etc.) and visual query languages, so giving them a semantics independent of the underlying data model.

The paper is organized as follows. In Section 2 the concepts of data model and representation are recalled; in Sections 3 and 4 the basic notions concerning the graph model and the Graphical Primitives are introduced. Section 5 shows how the most common data models and query languages can be represented in terms of the graph model and the Graphical Primitives. Finally, Section 6 concerns an example of formalization of a visual query language, namely that one of SNAP [Bryce Hull 1986], by means of the Graphical Primitives.

2.Preliminaries on Data Models and Associated Representations

A database and a query language are formally defined in terms of a data model and a set of operators respectively. A data model provides a set of structuring mechanisms that, in order to be perceived by a user, have to be expressed in terms of a *representation*. Several representations may be associated with a given data model. The same holds for a query language, whose operators have to be represented in some way in order to be used. However, in the past years the privileged component in the pair <model, representation> has been the model, while more recently, because of the growing interest in the field of the human-computer interaction, the two components are given equal importance (it may happen that the representation will be the prominent component in the future). One of the goals of our work is to reduce such a dichotomy and, consequently, the possible mismatch between the two components. Moreover, we aim to define a minimal set of graphical query operators general enough to be a formal basis for implementing, in principle, any complex VQL defined on a generic data model. Note that we restrict the information to be retrieved to the one explicitly represented in the database (i.e., we are not interested in inferring implicit information). In this case, the operations to be performed mainly consist in navigating through the data stored in the database and selecting or discarding some of them, based on either their mutual relationships and/or the comparison with constant values.

Starting from these considerations, we propose a minimal set of Graphical Primitives based on a graphical model, called graph model. The main characteristics of the graph model and the Graphical Primitives are:

a) the graph model allows us to define a graph model DataBase (*GMDB*) in terms of a triple $<g,c, m>$, where g is a graph-oriented structure, c is a set of constraints imposed on the classes of objects represented in g (g and c together constitute the intensional level of the database), and m (called interpretation) is a set-theoretical structure corresponding to the extensional level;
b) the semantics of the Graphical Primitives is characterized in terms of transformations, so that in the evaluation of a query, starting from the initial triple representing the database, a final triple is produced, containing exactly the requested information.

The Graphical Primitives of the language simply consist of two basic operations: *the selection of a node* and *the drawing of a labeled edge*. These two operations are used in all the three phases of query formulation, as introduced in Section 1, called *Location*, *Manipulation*, and *Visualization* phase. In the Location phase the user selects the part of the database s/he is interested in, by iteratively applying the operation of picking up a node. In the Manipulation phase the selected graph is modified, in order to restrict its interpretation. Referring to the relational algebra, it is possible to consider this phase as the one where operations such as selection, projection, joins, union, intersection, and difference are performed. The last phase, Visualization, has the aim of explicitly representing the structure of the query result, through the building of a new GMDB, whose interpretation consists only of the values determined according to the operations performed by the user in the previous phases. Note that the Manipulation phase can be used also after the Visualization phase for combining different subqueries, represented in terms of distinct GMDBs.

In order to be an adequate substratum for defining the Graphical Primitives, the graph model takes into account the most common features that all the existing data models exhibit, namely:

1. the notion of a *concept representing the common characteristics of a set of individuals*, which appears under different names in different models (e.g., entity in the entity-relationship model [Chen 1976], class in both semantic networks [Levesque Mylopoulos 1979]) and the object-oriented model[Kim 1990], and relation schema in the relational model [Codd 1972]);
2. the notion of *individual identity,* which is used to unambiguously distinguish the different individuals existing in the database (e.g., object identifier in the object-oriented model and key in both the Entity-Relationship and relational model);
3. the notion of *extension of a concept*, which provides the abstraction collecting individuals sharing common features (e.g., tuples in the relational model and instances in the entity-relationship model, in semantic networks, and in the object-oriented model);
4. the notion of *property* of a concept (e.g., attribute in both the Entity-Relationship and relational models);
5. the notion of *connections among concepts,* which allows for modeling the interrelationships existing in the real world (e.g., relationship in the entity-relationship model, link in both semantic networks and object-oriented model, and key duplication in the relational model [Hull King 1987]).

3. The graph model

In this section we give the formal definition of the concepts presented in Section 2. We first introduce the syntax and the semantics of the graph model in terms of *Typed Graph* and *Interpretation*, then define a suitable language for expressing *Constraints* on the elements of the Typed Graph.

The graph model allows us to define a GMDB *D* in terms of a triple $<g, c, m>$, where g is a Typed Graph, c is a set (possibly empty) of suitable constraints, and m is the corresponding Interpretation. The schema of a database, i.e., its intensional part, is represented in the graph model by the Typed Graph and the set of Constraints. The instances of a database, i.e., its extensional part, are represented by the notion of Interpretation.

Def. (Typed Graph)

A Typed Graph g is a tuple: $g= < N, E, \mathcal{L}_1, \mathcal{L}_2, f_1, f_2, f_3>$ where:

$N = N_C \cup N_R$ is the set of nodes; N_C is the set of so-called class-nodes, and N_R is the set of the so-called role-nodes. Moreover, N_C is partitioned into N_{C_p}, the set of printable class-nodes, and N_{C_n} , the set of unprintable class-nodes.

$E \subseteq N \times N$ is the set of edges;

$\mathcal{L}_1$ is the set of node labels;

$\mathcal{L}_2$ is the set of edge labels, including a special label T;

f_1 is a total biunivocal function from N to $\mathcal{L}_1$;

f_2 is a total function from E to $\mathcal{L}_2$;

f_3 is a total function mapping each node to one value in {*unselected, selected, displayed*}.

The labels in $\mathcal{L}_1$ are simply node names (i.e., names of both classes and roles), whereas the edge labels in $\mathcal{L}_2$ represent either set-oriented operations or boolean expressions, and are used in the process of query formulation (see Sections 4.1 and 4.2).

In the rest of the paper we use the following notations:

$AD\{n_1,...,n_k\}$ is the set of nodes adjacent to a given set of nodes $\{n_1,...,n_k\}$ minus $\{n_1,...,n_k\}$.

$\mathcal{D} = \mathcal{D}_p \cup \mathcal{D}_n$ is a set of elementary objects. $\mathcal{D}_p$ is a set of printable objects, $\mathcal{D}_n$ is a set of unprintable objets. Moreover, it holds that $\mathcal{D}_p \cap \mathcal{D}_n = \emptyset$.

$\mathcal{U}$ is a universe, that is a set of structured objects, defined as the smallest set containing $\mathcal{D}$ and all the possible labeled tuples (of any arity) $<l_1:t_1,...,l_n:t_n>$, where $l_1,...,l_n$ and $t_1,...,t_n$ are elements of $\mathcal{L}_1$ and $\mathcal{D}$, respectively.

Def. (Interpretation)

Let $g= < N, E, \mathcal{L}_1, \mathcal{L}_2, f_1, f_2, f_3>$ be a Typed Graph. An interpretation for g is a function $m: N \rightarrow 2^{\mathcal{U}}$ mapping each node $n \in N$ to a subset of $\mathcal{U}$ as follows:

if $n \in N_{Cp}$ then $m(n) \subseteq \mathcal{D}_p$;

if $n \in N_{Cn}$ then $m(n) \subseteq \mathcal{D}_n$;

if $n \in N_R$ and $\{n_1,...,n_h\} = AD\{n\} \cap N_C$, then $m(n)$ is a set of tuples of the form $<f_1(n_1):t_1,...,f_1(n_h):t_h>$, where $f_1(n_1),...,f_1(n_h) \in \mathcal{L}_1$ and $t_1,...,t_h \subseteq m(n_1) \times m(n_2) \times ... \times m(n_h)$.

In our model several types of constraints may be specified by means of a *Constraint Language*, which allows for representing the basic features of the most widely diffused models (as discussed in Section 2). In the following we concentrate on a suitable subset of the possible constraints, which may be graphically represented in a Typed Graph. The syntax of such constructs is as follows:

n_1 *ISA* n_2; where $n_1, n_2 \in N_C$;

ATLEAST(k, n_1, n_2); where $n_1 \in N_C$, $n_2 \in N_R$, and $k \in \mathbb{Z}$;

ATMOST(k, n_1, n_2); where $n_1 \in N_C$, $n_2 \in N_R$,and $k \in \mathbb{Z}$.

The corresponding semantics is:

n_1 ISA n_2 is satisfied by m if $m(n_1) \subseteq m(n_2)$;

ATLEAST(k,n_1,n_2) is satisfied by m if $\forall\ t^* \in m(n_1)\ \exists$ at least k tuples of the form $<l_1:t_1,...,f_1(n_1):t^*,.., l_p:t_p> \in m(n_2)$;

ATMOST(k,n_1,n_2) is satisfied by m if $\forall\ t^* \in m(n_1)\ \exists$ at most k tuples of the form $<l_1:t_1,...,f_1(n_1):t^*,.., l_p:t_p> \in m(n_2)$.

The ISA construct allows for representing subclass-class relationships; the ATLEAST and ATMOST constructs permit expressing cardinality constraints. Such constructs are graphically represented in a Typed Graph as shown in figure 1, i.e., a bold arrowhead edge for the ISA relationship, and a pair of numbers between brackets for the ATLEAST and ATMOST constructs. Moreover, we indicate with grayed circles the role nodes and with underlined labels the printable class nodes.

We note that, given a typed graph g and a set of constraints c, there always exists at least one interpretation m for g satisfying every constraint in c. Indeed, it is easy to verify that the interpretation mapping each node to the empty set satisfies every ISA and every cardinality constraint. However, it may happen that the constraints in c interact in such a way that some of the nodes in g are compelled to be invariantly empty in all the interpretations for g. Following [Lenzerini Nobili 1990], we say that g and c are strongly satisfiable if for each class node n_c of g there exists at least one interpretation m for g that satisfies c and such that $m(n_c)$ is not empty. A method for checking the strong satisfiability is studied in [Lenzerini Nobili 1990] in the context of a model that does not include ISA constraints. We are currently working on an extension of such a method in order to be able to check if a typed graph and a set of constraints are strongly satisfiable.

In the rest of the paper, we denote with *AD'{n}* the set defined as:

- if $n \in N_C$ then $AD'\{n\} = AD\{n\} \cup_i AD(n_i)$ where $n_i \in N_{Cn}$ and $n\ ISA^*\ n_i$ holds, where ISA* is the transitive closure of the ISA relation (note that if $n \in N_{Cp}$, then $AD'\{n\} \equiv AD\{n\}$);
- if $n \in N_R$ then $AD'\{n\} = AD\{n\} \cup \{m \in N_C \mid n \in AD'\{m\}\}$.

In other words, if n is a class node, the set AD'{n} contains both the nodes adjacent to n and the nodes adjacent to its ancestors in the ISA hierarchy; if n is a role node AD'{n} contains both the nodes adjacent to n and the descendants of such nodes.

An example of using the graph model, in order to model the information concerning persons, employees, and cars, is shown in Figure 1. Figure 1.a shows the Typed Graph g with the following constraints: Employee ISA Person; ATLEAST(1,Person,Age); ATMOST(1,Person,Age); ATLEAST(1,Car,Owns). Figure 1.b shows a possible Interpretation m for g.

4. Fundamental Graphical Primitives

In this section we formally define the Graphical Primitives (*GPs*). The main idea is to express any query-oriented user interaction with a database in terms of two simple graphical operations: the selection of a node and the drawing of a labeled edge. The former is the simplest graphical operation available to the user, and corresponds to switching the state of a node. The latter is the linkage of two nodes by a labeled edge, and corresponds to either restricting the node interpretations according to the rules stated in the label or performing a set operation on them. We show in the following that, by the composition of these simple mechanisms, all the phases of the query formulation (as introduced in Section 2) may be accomplished.

We assume that several views of a database may be used during query formulation. In order to build such views, we introduce the *DUPLICATE* function.

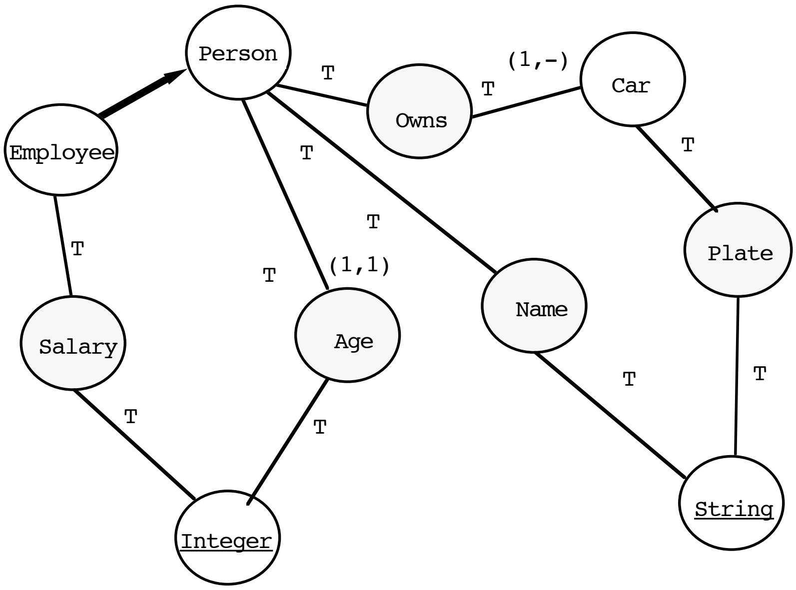


Figure 1.a

m (Employee) = {OI1, OI2, OI3};
m (Person) = {OI1, OI2, OI3, OI4};
m (Car) = {OI5, OI6};
m (Integer) = $\mathbb{Z}$;
m (String) = $\{a...z, 1...0\}^*$;
m (Age) = {<Person:OI1,Integer:22>, <Person:OI2,Integer:27>, <Person:OI3,Integer:18>, <Person:OI4,Integer:10>};
m (Name) = {<Person:OI1,String:Mary>, <Person:OI2,String:John>, <Person:OI3,String:Carl>, <Person:OI4,String:Cecily>};
m (Salary) = {<Employee:OI1,Integer:50.000>, <Employee:OI2,Integer:62.000>, <Employee:OI3,Integer:25.000>};
m (Plate) = {<Car:OI5,String:MaryNY123>, <Car:OI6,String:CA567>};
m (Owns) = {<Person:OI1,Car:OI5>, <Person:OI2,Car:OI6>}.

Figure 1.b

Figure 1: an example of use of the graph model.

The function *DUPLICATE^k(D)*, where D =<g,c,m> is a GMDB, results in a new GMDB D^k = $<g^k,c^k,m^k>$ (the k-copy of D) which is equal to D except for the node labels ($\circ$ denotes concatenation of two labels); in particular:

$N^k = N$;
$E^k = E$;
$\mathfrak{L}_1{}^k = \{k \circ l \mid l \in \mathfrak{L}_1\}$;
$\mathfrak{L}_2{}^k = \mathfrak{L}_2$;
$f_1{}^k = \{<n, k \circ f_1(n)>\}$;
$f_2{}^k = f_2$;
$f_3 k = f_3$;
$c^k = c$;
m^k is equal to m except for the labels of the tuple components of the role nodes:

$$m^k(n) = \{<k \circ l_1{:}t_1, \ldots, k \circ l_k{:}t_k> \mid <l_1{:}t_1, \ldots, l_k{:}t_k> \in m(n)\}$$

4.1 Selection of a Node and Drawing of an Edge

In the rest of the paper we denote with $D = <g,c,m>$ the database we operate on, and with $D' = <g',c',m'>$ the database resulting from the application of a GP.

Selection of a node n in D: $\mathcal{S}(D,n)$

$\mathcal{S}(D,n) = D'$ such that :
g' is equal to g except for $f_3'(n) = succ(f_3(n))$;
c' is equal to c;
m' is equal to m.

The succ function is defined on the domain of f_3 as follows: succ(unselected)=selected; succ(selected)=displayed; succ(displayed)=unselected.

The selection of a node is used to restrict the original graph g to a subgraph g'. We will see in Section 4.2 how this primitive well fit the Location phase of query formulation.

Drawing of a labeled edge in D: $\mathcal{E}(D,\mathfrak{F},n,q)$

This primitive can only be applied when no edge between n and q is in D. Its effect on the database D depends on the label $\mathfrak{F}$, which may be a boolean expression or a set-oriented operation.

Let n and q be role-nodes, and let $\mathfrak{F}$ be a boolean expression. The database $D'=\mathcal{E}(D,\mathfrak{F},n,q)$ is such that:
$N' = N$;
$E' = E \cup \{<n,q>\}$;
$\mathfrak{L}_1' = \mathfrak{L}_1$;
$\mathfrak{L}_2' = \mathfrak{L}_2 \cup \{\mathfrak{F}\}$;
$f_1' = f_1$;
$f_2' = f_2 \cup \{<<n,q>,\mathfrak{F}>\}$;
$f_3' = f_3$;
c' is equal to c;
$m' = m$.

D' differs from D only for the presence of a new labeled edge and the associate label. During the building of the result database D° (see Section 4.3) this will give rise to a restriction of the final interpretation.

Let n and q be unprintable class-nodes. The database $D'=\mathcal{E}(D,"=",n,q)$ is such that:
$N' = N$;
$E' = E \cup \{<n,q>\}$;
$\mathfrak{L}_1' = \mathfrak{L}_1$;
$\mathfrak{L}_2' = \mathfrak{L}_2 \cup \{=\}$;
$f_1' = f_1$;
$f_2' = f_2 \cup \{<<n,q>,=>\}$;
$f_3' = f_3$;
$c' = c$;

m' is equal to m except for m'(x), where $x \in AD(q)$: m'(x) = $\{<l_1:v_1,...,l_k:v_k,f_1(n):v_{k+1}> \mid <l_1:v_1,...,l_k:v_k,f_1(q):v_{k+1}> \in m(x)\}$.

Note that this operation corresponds to renaming of a tuple component in all the adjacents of a node q. It is useful for handling queries involving more than once the same node, each occurrence belonging to a different user view of the database.

Finally, if $\mathfrak{F}$ is a set operator, say setop (union, difference, and intersection), then n and q are class nodes and D' will contain both a new node and new ISA constraints. More precisely, D'=$\mathfrak{C}$(D,$\mathfrak{F}$,n,q) is such that:
$N_C' = N_C \cup \{s\}$ (s is a new class node);
E'= E ;
$\mathfrak{L}_1' = \mathfrak{L}_1 \cup \{et_s\}$, where et_s is a new label for s;
$\mathfrak{L}_2' = \mathfrak{L}_2$;
$f_1' = f_1 \cup \{<s,et_s>\}$;
$f_2' = f_2$;
$f_3' = f_3 \cup \{<s,displayed>\}$;
c' is equal to c plus new ISA constraints concerning n, q and s, namely:
if $\mathfrak{F} = \cup$ then c'= c $\cup$ {n ISA s, q ISA s};
if $\mathfrak{F} = \cap$ then c'= c $\cup$ {s ISA n, s ISA q};
if $\mathfrak{F} = -$ then c'= c $\cup$ {s ISA n};
m' is equal to m except for m'(s): m'(s) = m(n) setop m(q);

We will see in the following that drawing a labeled edge is the basic mechanism in the Manipulation phase.

The above two GPs constitute the minimal set of elementary interactions. However, for the sake of simplicity, we add a further operation, namely the change of label of an edge linking a class-node s to a role-node q. The change of an edge label is denoted with $\mathfrak{C}(D,\mathfrak{F},s,q)$, where $\mathfrak{F}$ is a propositional formula whose atoms are of the form $\alpha R \beta$, where R is a comparison operator; α and β are either the s component of the tuples belonging to the interpretation of q (referred through the label of q) or constants. During the building of the result database D° (see Section 4.3) the presence of labels different from the true value T will give rise to a restriction of the final interpretation. It can be shown that the same result of the operation of changing a label can be obtained by drawing a reflexive edge on the role-node q. It follows that the label changing is not strictly necessary.

4.2 The query phases

In this subsection we deal with the three basic interaction phases, namely Location, Manipulation, and Visualization, according to the structure of query formulation as illustrated in Section 2.

In the Location phase the user may build several views of the same database, each one suitable to the formulation of a query. Such queries can be then composed in the Manipulation phase, giving rise to an integrated result. The visual mechanism used in the Location phase is the selection of a node: by means of a sequence of selections the user may locate the whole subgraph of interest. Note that such a subgraph may be disconnected. Moreover, the user cannot select two or more different class nodes at the same level of a ISA hierarchy. For selecting the two above nodes, two or more views of the initial database have to be built by means of the DUPLICATE function, then a different node is selected in each view. In practical systems, the Location phase may be accomplished by more sophisticated strategies, always based on the idea of changing the state of a set of nodes. For instance, a system could provide an algorithm that searches all the existing paths connecting two selected nodes; such paths are shown to the user that decides whether to include them in the schema of interest. Nevertheless, while the available strategies may be very sophisticated and different, their effects can always be expressed in terms of node selections.

The Manipulation phase is mainly based on changing the label of an existing edge and/or drawing a new labeled edge (whose label is a boolean formula) between two role-nodes. The goal of the Manipulation phase is in specifying the conditions the result of the query has to satisfy. This is obtained by modifying the drawing of the Typed Graph, as resulting from the Location phase. Moreover, it can be used in order to compose several subqueries, by drawing an edge labeled with a set operator between two class nodes. For example, if the user is interested in the query "List the employees with a salary greater than thirty thousand dollars or owning a car that costs more than twenty thousand dollars," s/he may perform two separate subqueries (the first one referring to the salaries, the second one to the cars), and then draw an edge, labeled with the union operator, between the two class nodes representing the partial results.

The Manipulation phase is the closest phase to the textual query formulation, as it is carried on in traditional systems. At the end of this phase the user has fully specified the query structure, and the only thing that is missing is computing the query result and representing it. This is done in the Visualization phase, where the structure of the query result is visualized as a new GMDB, the *result database* D°. The new typed graph is constituted by a single non-printable class node (denoted with q in the following), a set of role-nodes (denoted with $\{r_1,\ldots,r_k\}$, whose labels are the ones of the "displayed" role nodes of D), and a set of printable class nodes of D, linked with the above role nodes. The result database is mainly related to the idea of displaying the structure of the query result homogeneously with the input database and to possibly reuse it (i.e., the model is closed under the query operations). The process of constructing the result database is detailed in the following.

4.3 Result Database and Admissible GMDB

We have seen in the previous subsection that the product of the location and the manipulation phase is again a GMDB. We now describe how to associate with such a GMDB D a new GMDB, called D°, denoting the information content of the query represented by D. We also provide necessary and sufficient conditions on D in order for D° to contain a non empty set of nodes, i.e., to be admissible. Roughly speaking, D° is a GMDB composed by a unique unprintable class node linked, by means of binary role nodes, to a set of printable nodes, corresponding to the ones set to displayed in D. The interpretation of the above binary role nodes is computed in two logical steps: in the first step all the selected role nodes of D are joined together giving the meaning of a fictitious n-ary role node; in the second step such a meaning is suitably projected on the binary role nodes of D°.

Let D be a GMDB. Let $r_1,\ldots,r_k \in N_R$ be the role nodes of D such that $f_3(r_i)$= displayed and there exists $m \in AD(r_i)$ with $m \in N_{Cp}$ and $f_3(m)$=displayed. Let $m_1,\ldots,m_h \in N_{Cp}$ the printable class nodes of D such that $f_3(m_i)$=displayed and there exists $r \in AD(m_i)$ with $r \in N_R$ and $f_3(r)$=displayed.

Let us denote with *new*(H_i), where $H_i \in \{r_1,\ldots,r_k, m_1,\ldots,m_h\}$ a function associating with H_i one invented node, such that $new(H_i) \neq new(H_j)$ for $i \neq j$. Finally, let q be a new node, and let et_q be a new node label.

Let D° be the result database associated with D. The typed graph of D° is defined as follows:

$N_R{}^\circ$ = {new(r_1), .., new(r_k)};

$N_{Cn}{}^\circ$ = {q};

$N_{Cp}{}^\circ$= {new(m_1), .., new(m_h)};

$E^\circ=\{<new(m_i),new(r_j)> \mid <m_i,r_j> \in E\} \cup \{<q,new(r_j)> \mid r_j \in \{r_1,..,r_k\}\}$;

$\mathfrak{L}_1{}^\circ=\{et_q\} \cup \{f_1(r_i) \mid r_i \in \{r_1,..,r_k\}\} \cup \{f_1(m_i) \mid m_i \in \{m_1,..,m_h\}\}$;

$\mathfrak{L}_2{}^\circ = \{T\}$;

$f_1{}^\circ=\{<q,et_q>\} \cup \{<new(x),et> \mid <x,et> \in f_1 \text{ and } x \in \{r_1,...,r_k, m_1,...,m_h\}\}$;

$f_2{}^\circ=\{<e,T> \mid e \in E^\circ\}$;

$f_3{}^\circ=\{<x,\text{unselected}> \mid x \in N^\circ\}$;

$c^\circ = \emptyset$;

In order to specify m°, i.e., the interpretation of q, $r_1,...,r_k$, we need to introduce some useful functions and to characterize some intermediate results (i.e., *RIS* and *RIS'*).

Let *HSN(x)* be a function defined on a class node x and returning a true value if x is the highest selected node in the ISA hierarchy it belongs to (i.e., is x the highest selected node?): HSN(x) = True iff f_3 (x) ∈ {selected, displayed} and there exists no w ∈ N_C such that x ISA* w and w ∈ {selected, displayed}.

Let *TAD(y)* (i.e., the set of true adjacents of *y*) be a function defined on a role node *y* and returning the subset of AD'(y) satisfying the HSN condition: TAD(y) = {x | x ∈ AD'(y) and HSN(x)}.

Let *RM(m(y))* (i.e., restrict the meaning of y) be a function defined on the interpretation of a role node y, which results into an interpretation of y, restricted according to the selected class nodes in the ISA hierarchies which are in TAD(y). As a consequence, this function suitably changes the labels of the tuples in the interpretation of y as well.

$RM(m(y)) = \{<l_1':v_1, ..., l_k':v_k> \mid <l_1:v_1, ..., l_k:v_k> \in m(y)$ and $l_i':v_i$ are such that $l_i'=f_1(TAD(y) \cap \{z \in N_C \mid z \text{ ISA* } f_1{}^{-1}(l_i)\}$ and $v_i \in m(f_1{}^{-1}(l_i'))$).

Let *RM'(m(y))* be a function defined on the interpretation of a role node y, which, in order to compute the interpretation of D°, concatenates the label of y to the labels of the tuples in the interpretation of y (∘ denotes concatenation of two labels):

$RM'(m(y)) = \{<l_1'':v_1, ..., l_k'':v_k> \mid <l_1':v_1, ..., l_k':v_k> \in RM(m(y)) \text{ and } l_i''=f_1(y) \circ l_i'\}$.

On the basis of the above functions we can define the interpretation of D° as follows.

$m^\circ(q) = \{t_1,...,t_s\}$, where each t_i is a new invented unprintable value, and s is the cardinality of a set RIS that can be interpreted as the extensional part of the user query, and is defined as follows:

- if $N_R{}^\circ = \emptyset$ then RIS = ∅;
- if $|N_R{}^\circ|=1$ and $|\{k \in N_R \mid f_3(k) \in \{\text{selected},\text{displayed}\}\}|=1$ then RIS=RM'(m(k)), where $k \in N_R$ and $f_3(k) \in \{\text{selected},\text{displayed}\}$;

- otherwise, $N_R^\circ = \{r_1,..,r_k\}$, with $k \geq 1$ and $RIS = inst(eval(n_1, eval(n_2,..,eval(n_{h-1}, n_h))))$, where $\{n_1,..,n_h\} \equiv \{m \in N_R \mid f_3(m) \in \{selected,displayed\}\}$ and the function inst extracts the set of instances of a fictitious node computed by eval.

Eval(n_1,n_2) returns a fictitious role node n, whose adjacents are the union of the adjacents of n_1 and n_2, i.e., $AD(n) = AD(n_1) \cup AD(n_2)$, and whose interpretation is a set of tuples:

$\{<l_1:v_1,..,l_k:v_k,\ "nc" \circ l_{k+1}:v_{k+1},..,"nc" \circ l_h:v_h,\ l_{h+1}:v_{h+1},..,l_t:v_t>\}$

such that for $i=k+1..h$, $f_1^{-1}(l_i) \in N_{Cn}$ and

$<l_1:v_1,..,l_k:v_k,\ f_1(n_1) \circ l_{k+1}:v_{k+1},..,f_1(n_1) \circ l_h:v_h> \in RM'(m(n_1))$

$<f_1(n_2) \circ l_{k+1}:v_{k+1},..,f_1(n_2) \circ l_h:v_h,\ l_{h+1}:v_{h+1},..,l_t:v_t> \in RM'(m(n_2))$.

Note that, if the nodes n_1 and n_2 do not share tuple components, the function eval returns the cartesian product of the interpretations of n_1 and n_2.

Let us denote with *RIS'* the set of tuples obtained by restricting the set of tuples of RIS to the ones satisfying all the boolean expressions $\mathfrak{F}_1, ..,\mathfrak{F}_n$ labeling the edges of D, and by adding to the remaining tuples of RIS a new component, that is:

RIS' is the set of tuples of the form $<l_1:v_1,...,l_y:v_y,\ f_1(q):v_{y+1}>$ such that $v_{y+1} \in m^\circ(q)$, $<l_1:v_1,...,l_y:v_y> \in RIS$ and satisfies $\mathfrak{F}_1, ..,\mathfrak{F}_n$, such that different tuples have different values in the y+1-th component.

We finally obtain from RIS' the interpretation of the role-nodes in D°:

$m^\circ(r_i) = \{<l:v,\ f_1(q):v_{y+1}> \mid <l_1:v_1,...,f_1(r_i) \circ l:v,...,l_y:v_y,\ f_1(q):v_{y+1}> \in RIS'\}$.

Note that, depending on the structure of the GMDB D, the result database D° may have an empty set of role nodes. In this case we say that D is not admissible. The following lemma specifies the constraints a GMDB must satisfy in order to be admissible.

Lemma 1

A GMDB D is admissible iff the following conditions hold:

1) it exists a role node n in D such that $f_3(n)$=displayed;
2) for each role node n in D such that $f_3(n)$=displayed it holds that:

   $|AD(n) \cap \{k \in N_C \mid f_3(k) \in \{selected,displayed\}\}| \geq 2$ and

   $|AD(n) \cap \{k \in N_{Cp} \mid f_3(k)=\{displayed\}\}| \geq 1$.

In other words, in order to be admissible, a GMDB must contain at least one displayed role node and each displayed role node must be linked to at least one displayed printable class node and one unprintable (selected or displayed) class node.

4.4 Example

As we said in the introduction, the aim of this paper is mainly in proposing a new approach to the VQLs formalization, based on the definition of a minimal set of GPs. We think that such primitives have to be essentially used as a formal basis for building more powerful and friendly visual query operators, working on databases expressed in terms of the most common data models. Nevertheless, the GPs may be directly used for querying the GMDBs, as shown in the following example.

Referring to the GMDB in figure 1, we are interested in retrieving the names of all the persons whose age is greater than 21 and owning a car with plate containing the person name. We assume that the GMDB D is such that:

1. $f_3(n)$ = unselected for each $n \in N$;
2. $f_2(<n,q>) = T$ for each $<n,q> \in E$;
3. $E \cap (R \times R) = \emptyset$.

According to the above querying phases, we describe our interaction in terms of:

- Location phase:
  - the nodes *Person*, *Age*, *Owns*, *Car*, *Plate*, and *Integer* are selected, while the nodes *Name* and *String* are displayed, giving rise to the view shown in Fig. 2.a.
- Manipulation phase:
  - the label of the edge <age, integer> is changed into *Age.Integer > 21*;
- an edge, characterized by the label *Name.String* IN *Plate.String*, is drawn between the nodes *Name* and *Plate* (IN is a boolean operator checking whether one string is a substring of another one). The resulting Typed Graph is shown in Fig. 2.b.
- Visualization phase:
  - the system evaluates the result database and displays the corresponding GMDB (see Fig.2.c). The only individual satisfying the conditions specified in the above phases is OI1, having *Name* "Mary", *Age* 22, and *Car Plate* "MaryNY123". This results into creating a new object OI20, constituting the interpretation of the node *RESULT*, with value "Mary" for the relationship *Name*.

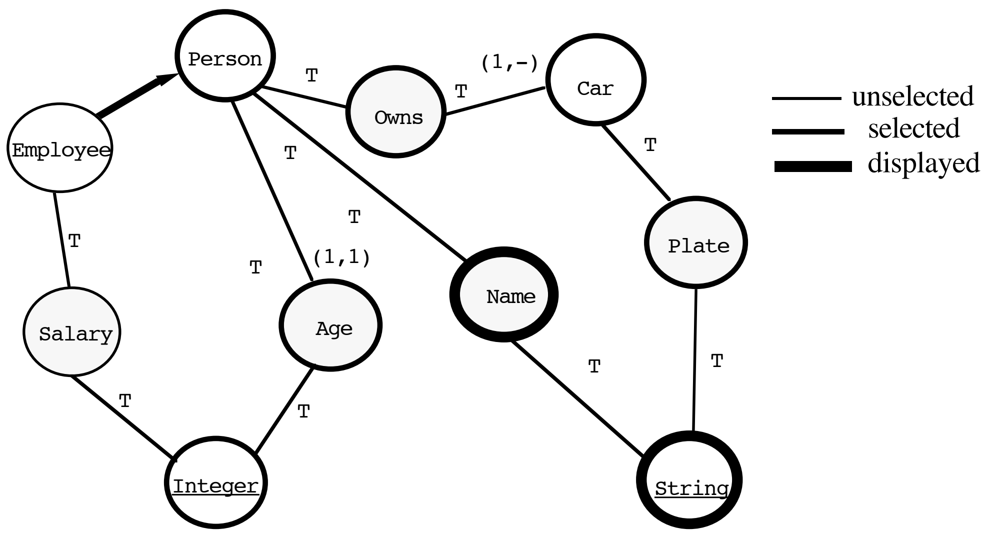


Figure 2.a

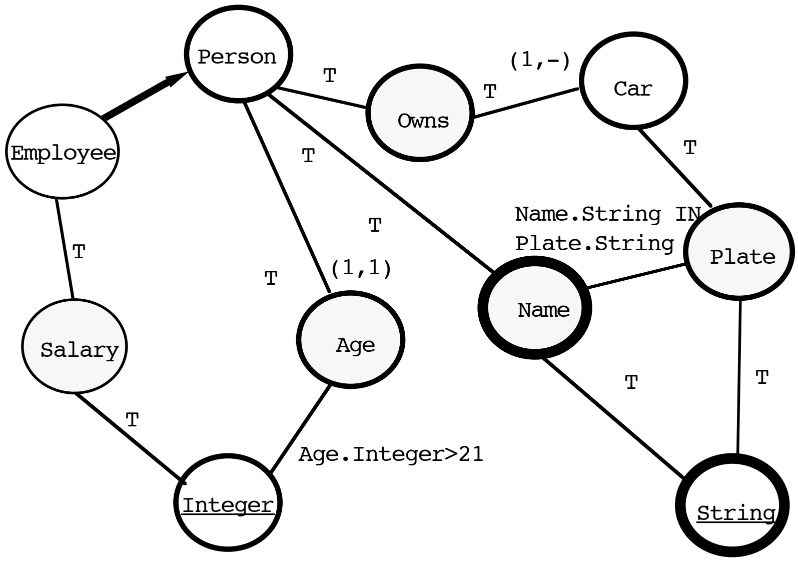
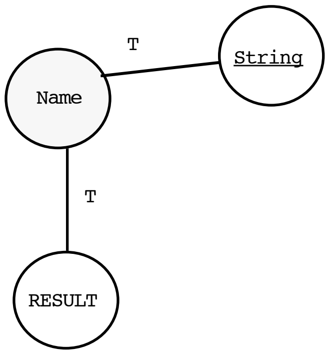


Figure 2.b



m (RESULT) = {OI20};
m (String) = {a...z, 1...0}*;
m (Name) = {<Person:OI20,String:Mary>};

Figure 2.c

Figure 2: An example of query formulation

5. Applications to Value-oriented and Object-oriented Data Models

The generality of our approach allows the representation of a wide class of models, such as relational, semantic, and object-oriented models, in terms of the graph model constructs. Moreover, the GPs defined on the graph model can be effectively used for characterizing the semantics of the most popular visual query languages defined on both semantic and object-oriented data models (a comprehensive survey of such languages is in [Batini et al. 1991]). In principle, the graph model can be used as the internal model of any visual query system, providing a sound formalism useful to characterize the semantics of the visual operations defined on the external model.

As we said in the introduction, our proposal differs from others available in the literature mainly for the problems it intends to solve. Perhaps, the works that are most similar to the ours are [Mark 1989] and [Gyssens Paredaens Van Gucht 1990]. Both the papers present new VQSs, which are intended to be directly utilized by the users. Our proposal is mainly in defining a tool for VQL formalization by introducing a minimal set of simple querying primitives, which are intrinsically graphical and constitute the basis for defining more complex operations. Our proposal shares several similarities with Mark's approach both in the data model adopted(except for the arity of relationships, which are only binary in Mark's work, while our role-nodes can be linked to n other nodes) and in the way of computing the query result, while the process of query formulation is deeply different in that keywords instead of graphical operations are basically used. In [Gyssens Paredaens Van Gucht 1990] a powerful object-oriented data model is defined, called GOOD, provided with a graphical interface and graphical interaction for all the typical database operations. Besides distinct motivations, this approach is different from ours from a graphical point of view. In GOOD, a query is specified by a pattern that is matched against the instances graph, while our GPs work on the intensional level of the database and their semantics is defined in terms of set operations. Finally, the higher expressive power of the GOOD query language is obtained by introducing methods, which are strictly related to the object-orientation paradigm and are not expressible in terms of graphical operations.

We now consider two notable classes of data models, namely the value-oriented and the object-oriented models, representative of the two main modeling styles introduced in the literature (see [Ullman 1987]) and specify the rules they induce on the graph model.

5.1 Value-Oriented Models

Among the value-oriented models, the relational model [Codd 1970] is the most representative one, since it has the simplest and the most uniform data structures and is the most formal in nature. The relational model represents data as a collection of relations. Informally, each relation resembles a table, and each row in the table represents a collection of related data values. More formally, a relational database can be defined as follows.

Let $\mathtt{DN} = \{\mathtt{dn}_1,..,\mathtt{dn}_n\}$ be a set of *domain names*, and let $\mathtt{D} = \{\mathtt{D}_1,..,\mathtt{D}_n\}$ be a family of *domains*, each composed by a set of *values* (we suppose to have finite domains); let $\mathtt{eval}$ be a function, $\mathtt{eval} : \mathtt{DN} \rightarrow \mathtt{D}$, associating a domain to each name; let $\mathtt{A} = \{\mathtt{a}_1,..,\mathtt{a}_m\}$ be the *set of attribute names*, and let $\mathtt{dom} : \mathtt{A} \rightarrow \mathtt{D}$ be a function which associates a domain $\mathtt{dom(a)}$ to each $\mathtt{a} \in \mathtt{A}$. Let $\mathtt{RN} = \{\mathtt{rn}_1,..,\mathtt{rn}_k\}$ be a set of *relation names*, and let $\mathtt{attr}: \mathtt{RN} \rightarrow 2^{\mathtt{A}}$ be a function which associates to each $\mathtt{rn} \in \mathtt{RN}$ a non empty set $\mathtt{attr(rn)}$ of attributes, called *relation schema*, and let $\mathtt{inst}$ be a function, which associates to each $\mathtt{rn} \in \mathtt{RN}$ a finite set $\mathtt{inst(rn)}$, called *relation instance* and composed of tuples of the form $<\mathtt{a}_1{:}\mathtt{v}_1, \mathtt{a}_2{:}\mathtt{v}_2,..,\mathtt{a}_h{:}\mathtt{v}_h>$ where $\{\mathtt{a}_1,..,\mathtt{a}_h\} = \mathtt{attr(rn)}$, $\mathtt{v}_i \in \mathtt{dom(a}_i\mathtt{)}$. A *relation* is a triple $\mathtt{r} = <\mathtt{rn}, \mathtt{attr(rn)}, \mathtt{inst(rn)}>$, with $\mathtt{rn} \in \mathtt{RN}$. A *relational database* $\mathtt{DB} = \{\mathtt{r}_1,..,\mathtt{r}_k\}$ is a set of relations with different names.

In order to represent the relational model in terms of the graph model, we need first of all to introduce a set of rules restricting the generality of our model, then to define a precise mapping between the two.

A *relational graph model database* is a GMDB that satisfies the following constraints:

- for each node n in N_R it holds that:

  $|AD(n) \cap N_{Cn}| = 1$;
  $|AD(n)| = 2$.
- for each edge $<n,q>$, where $n \in N_R$ and $q \in N_{Cn}$, the constraints ATLEAST(1, n, q) and ATMOST(1, n, q) are satisfied.

In other words, each role node represents a function linking a non-printable class node to a printable class node. This reflects the structure of a relational schema, in which the non-printable class node corresponds to the relation name, the role nodes to its single-valued attributes, and the printable class nodes to attribute domains. No link is allowed between different relations, so that unprintable class nodes cannot be related via a role node. The interpretation of each non-printable class-node is made of a set of special values, that do not belong to D and may be considered as tuple identifiers (see pp.352-354 in [Ullman 1987]). We assume the availability of a function, called `new`, that associates to each tuple a (unique) non-printable value.

We denote with $\mathcal{H}$ the mapping from a relational database to the corresponding database expressed in terms of the graph model. If DB is any relational database, $\mathcal{H}(\mathtt{DB}) = D = \langle g, c, m\rangle$ is defined as follows:

1) For each $\mathtt{dn}_i \in \mathtt{DN}$ and $\mathtt{D}_i = \mathtt{values}(\mathtt{dn}_i)$, there is a corresponding node $n_i \in N_{Cp}$ such that $f_1(n_i) = \mathtt{dn_i}$ and $m(n_i) = \mathtt{D}_i$;
2) For each $\mathtt{rn}_i \in \mathtt{RN}$ there is a corresponding node $n_i \in N_{Cn}$ such that $f_1(n_i)=\mathtt{rn_i}$and its interpretation coincides with a set of new values generated by applying the `new` function to all the tuples belonging to $\mathtt{inst}(\mathtt{rn}_i)$:$m(n_i)=\{new(t)\}$, $t \in \mathtt{inst}(\mathtt{rn}_i)$;moreover, for each $\mathtt{a} \in \mathtt{attr}(\mathtt{rn}_i)$, whose corresponding node is n_j (see 3), there is the edge $\langle n_i,n_j\rangle$ in D;
3) For each $\mathtt{a} \in \mathtt{A}$ there is a corresponding node $n_j \in N_R$ such that $f_1(n_j) = \mathtt{a}$ and there is an edge $\langle n_j, n_k\rangle$, where $n_k \in N_{Cp}$ is the node corresponding to $\mathtt{dom(a)}$.Moreover, $m(n_j) = \{\langle \mathtt{a}{:}x,\mathtt{b}{:}y\rangle \mid$ there exists a tuple t in the relation $\mathtt{b}$, such that $y=new(t)$ and x is the $\mathtt{a}$-component of t$\}$.

Note that D is a relational GMDB. Conversely, we define a mapping $\mathcal{H}'$ from any relational GMDB to the corresponding relational database as follows:

1) For each $n_i \in N_{Cp}$ there is a corresponding domain $\mathtt{D}_i=m(n_i)$, with name $\mathtt{dn_i} = f_1(n_i)$;
2) For each $n_i \in N_R$ there is a corresponding attribute $\mathtt{a_i} = f_1(n_i)$, $\mathtt{dom}(\mathtt{a}_i) = m(AD(n_i) \cap N_{Cp})$;
3) For each $n_i \in N_{Cn}$ there is a corresponding relation name $\mathtt{rn_i}= f_1(n_i)$, such that $\mathtt{attr}(\mathtt{rn}_i) = \{f_1(n_j)\}\ \forall n_j \in AD(n_i)$,$\mathtt{inst}(\mathtt{rn}_i)=\{\langle f_1(n_1){:}c_1,f_1(n_2){:}c_2,\ldots,f_1(n_h){:}c_h\rangle \mid \langle f_1(n_j){:}c_j,f_1(n_i){:}c^*\rangle \in m(n_j)$, where $\{n_1,\ldots,n_h\} = AD(n_i)$, and $c^* \in m(n_i)\}$;

Notice that if DB is a relational database, then $\mathcal{H}'(\mathcal{H}(\mathtt{DB})) = \mathtt{DB}$.

PERSON

Age	Name
22	Mary
27	John
18	Carl
10	Cecily

OWNS

P.Name	C.Plate
Mary	NY123
John	CA567

CAR

Plate
NY123
CA567

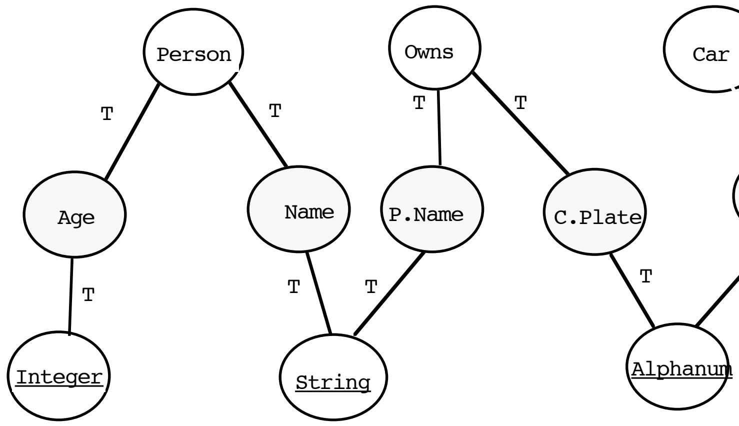


Figure 3.a

m (Person) = {OI1, OI2, OI3, OI4};
m (Car) = {OI5, OI6};
m (Owns) = {OI7, OI8};
m (Integer) = $\mathbb{Z}$;
m (String) = $\{a...z\}^*$;
m (Alphanum) = $\{a...z, 1..0\}^*$;
m (Age) = {<Person:OI1,Integer:22>, <Person:OI2,Integer:27>, <Person:OI3,Integer:18>, <Person:OI4,Integer:10>};
m (Name) = {<Person:OI1,String:Mary>, <Person:OI2,String:John>, <Person:OI3,String:Carl>, <Person:OI4,String:Cecily>};
m (P.Name) = {<Owns:OI7,String:Mary>, <Owns:OI8,String:John>};
m (Plate) = {<Car:OI5,String:NY123>, <Car:OI6,String:CA567>};
m (C.Plate) = {<Owns:OI7,String:NY123>, <Owns:OI8,String:CA567>};

Figure 3.b

Figure 3: A relational database and the corresponding one in the graph model

Rule 1 states that the family of domains `D` corresponds to the set of interpretations of the printable nodes of g (i.e., N_{Cp}) and the names of such domains are given by the labels of the elements of N_{Cp}. Rule 2 states that the set `A` of attribute names coincides with the set of labels of the role nodes (i.e., N_R) of g, and that the associated domains correspond to the interpretation of the adjacent printable nodes (note that each role node has a single adjacent printable node as said above). Rule 3 states that the set `RN` of relation names coincides with the set of labels of non-printable nodes in g, while, for each `rn`, the attribute names `attr(rn)` coincide with the set of labels of the adjacent role nodes. Moreover, the instance set of each relation `rn` corresponds to the set of tuples obtained by joining the interpretations of the adjacent role nodes sharing the same tuple component coming from the interpretation of the non-printable node corresponding to `rn`. Finally, each relation `r` corresponds to a triple whose first element is a non-printable node label, the second element is the set of labels corresponding to its adjacent role nodes, the third element is the corresponding tuple set as defined above. In figure 3 an example of mapping between relational and GMDBs is drawn.

5.2 Relational completeness

In this subsection we deal with the expressive power of the GPs. We prove that the class of queries computed by the GPs (denoted as Q(GPs)) contains the class of queries computable by the

Relational Algebra (denoted as $Q(-,\cup,\ \times,\ \Pi_A,\ \sigma_F)$, see [Ullman 1987]). Note that D° denotes the result database associated with D, as introduced in Section 4.3.

Lemma 2

Let DB be a relational database, made of a single relation r = <rn, attr(rn), inst(rn)>, and let D the GMDB obtained by setting to "displayed" all the nodes in $\mathcal{H}$(DB). If the label of the unprintable node q of D° coincides with rn then $D^\circ \equiv D$, and D° is a relational GMDB.

What this lemma says is that we can always choose the label of the node q in D° in such a way that $D^\circ = \mathcal{H}$(DB). In what follows we assume that the label of q in D° is always chosen so as to satisfy the above condition.

Theorem 1

$Q(-, \cup, \times, \Pi_A, \sigma_F) \subseteq Q(GPs)$.

Proof

We show that given any relational database DB and any query in $Q(-, \cup, \times, \Pi_A, \sigma_F)$, with result R on DB, we can always find a query Z expressed in terms of the GPs, such that the result of Z to $\mathcal{H}$(DB) is T and $\mathcal{H}'$(T)= R.

We proceed by induction on the number of occurrences of operators in the relational algebra expression.

The basis is i=0, that is a single operand R in the relational algebra expression. By definition and according to Lemma 2, we have a database D such that $\mathcal{H}'(D^\circ)$=DB, where DB is the database made by the single relation R.

Consider now an expression whose outermost operator is one of union, difference, projection, selection, and product.

Case 1: The expression is $E=E_1\cup E_2$. Let $D_1 = \mathcal{H}(E_1)$ and $D_2 = \mathcal{H}(E_2)$. By the inductive hypothesis, $\mathcal{H}'(D^\circ_1)=E_1$ and $\mathcal{H}'(D^\circ_2)=E_2$. Then, we apply the GP $\mathcal{E}$, in particular for drawing a labeled edge between $q^\circ_1 \in D^\circ_1$ and $q^\circ_2 \in D^\circ_2$, whose label is the set union operator. The resulting database is $D_3=\mathcal{E}(D_1+D_2,\cup,q^\circ_1,q^\circ_2)$, where + means simply taking into account simultaneously two separate databases instead of one. Under this conditions, it holds that: $\mathcal{H}'(D^\circ_3)$=DB.

Case 2: The expression is $E=E_1-E_2$. Following the same reasoning line as in case 1, with the new edge labeled with the minus set operator, we have: $\mathcal{H}'(D^\circ_3)$=DB.

Case 3: $E=\Pi_A(E_1)$. Let $D_1 = \mathcal{H}(E_1)$, by the inductive hypothesis we have $\mathcal{H}'(D^\circ_1)=E_1$. Then, we select all the nodes in D_1 except for the role nodes whose labels belong to $(attr(E_1)-A)$ obtaining $D_3 = \mathcal{S}(D_1,n_1)\circ\mathcal{S}(D_1,n_2)\circ\ldots\circ\mathcal{S}(D_1, n_h)$, where each n_i is such that $f_1(n_i) \in \mathcal{L}_1-(attr(E_1)-A)$. It is easy to see that $\mathcal{H}'(D^\circ_3)$=DB.

Case 4: $E=\sigma_F(E_1)$. Let $D_1 = \mathfrak{H}(E_1)$. Following Ullman (Lemma 3.5 in [Ullman 1987]), we may assume that F is a simple selection. By the inductive hypothesis we have $\mathfrak{H}'(D^\circ_1)=E_1$. Then, simply by substituting with F the label of the edge linking the role node n_i (i.e., the node whose label coincides with the attribute name involved in F itself) to the printable node n_j (i.e., the node corresponding to the attribute domain), we obtain $D_3=\mathfrak{C}(D_1,F,n_i,n_j)$. Now, it is easy to see that $\mathfrak{H}'(D^\circ_3)=DB$.

Case 5: $E=E_1 \times E_2$. Let $D_1 = \mathfrak{H}(E_1)$ and $D_2 = \mathfrak{H}(E_2)$. By the inductive hypothesis $\mathfrak{H}'(D^\circ_1)=E_1$ and $\mathfrak{H}'(D^\circ_2)=E_2$. We select all the nodes of D_1 and D_2, it holds that $D_3=\mathcal{S}(D_1, n_1)\circ\mathcal{S}(D_1, n_2)\circ..\mathcal{S}(D_1), n_h)\circ\mathcal{S}(D_2, n_1)\circ\mathcal{S}(D_2, n_2)\circ...\mathcal{S}(D_2, n_k)$, where each n_i belongs to either D_1 or D_2, and $\mathfrak{H}'(D^\circ_3)=DB$. Æ

The above result shows that the GPs, applied on relational GMDBs, allows us to express, as the relational algebra do, at least the first-order queries.

5.3 Object-oriented and Semantic Data Models

Both object-oriented and semantic data models are equipped with powerful structural abstractions (e.g., generalization, classification, aggregation) that satisfy the representation requirements of the new kinds of applications (CAD, CAM, AI). The difference between semantic and object-oriented models is, essentially, that semantic models describe structural aspects of objects, whereas object-oriented models also encapsulate behavioral aspects [Hull King 1989]. Anyhow, it is generally accepted that encapsulation can be violated in ad-hoc query mode for permitting free associative access to data (see, e.g., [Bancilhon Cluet Delobel 1990] for a discussion of this topic). For such a reason, in this paper we focus only on structural features, which represent the common aspect of the two classes of data models. Moreover, even though in most of the object-oriented proposals the elementary domains are considered as *primitive classes* (i.e., classes with no attributes) in the following we will distinguish between classes and domains.

No general agreement exists on the definition of the above data models. Anyhow a number of concepts have been identified as the salient features of the approach, e.g., object identity, encapsulation, class, and inheritance (see, e.g., [Bancilhon 1988; Kim 1990; Beeri 1990; Cruz 1990]).

An *object-oriented database* (OODB) is a collection of objects, classes, and elementary domains. An *object* of the database corresponds to an object in the real world, and exists regardless of the value of its properties. In other words, each object has an *identity*, different from that of any other object, that does not change throughout its lifetime. Each object has a unique 4identifier, called *object identifier (oid)*, which distinguishes it from all the others. Every object encapsulates a *structure*, that is a set of relationships (called *attributes* in the following) with both other objects and values in elementary domains, which may be one-to-one or one-to-many. Objects that share the same set of attributes may be grouped into a *class*. Because of no limitation exists on the domain of attributes, i.e., an attribute domain can range on another class, the definition of a class may results in a nested structure. The relationship between an object and its class is the well-known *instance-of* relationship. Classes may be related through a generalization relationship, usually called *is-a* relationship; a class may have any number of *sub-classes* and *super-classes*. A class inherits the attributes of its superclasses (conflicts possibly caused by multiple inheritance are generally handled by the system) and may have additional attributes. An object belongs to one class, as an *instance* of that class; furthermore, it is instance also of all the superclasses of the class it belongs to.

Summarizing, let $\mathtt{CN} = \{\mathtt{cn}_1,\ldots,\mathtt{cn}_n\}$ be the set of class names, $\mathtt{DN} = \{\mathtt{dn}_1,\ldots,\mathtt{dn}_n\}$ be the set of *domain names*, and let $\mathtt{D} = \{\mathtt{D}_1,\ldots,\mathtt{D}_n\}$ be the family of *domains*, each composed by a set of *values* (we suppose to have finite domains); let `values` be the function, $\mathtt{values}: \mathtt{DN} \rightarrow \mathtt{D}$, associating a name to each domain; $\mathtt{OID} = \{\mathtt{oid}_1,\ldots,\mathtt{oid}_k\}$ be the set of object identifiers, and $\mathtt{AN} = \{\mathtt{an}_1,\ldots,\mathtt{an}_p\}$ be the set of attribute names. A class `C` is a 4-tuple $<\mathtt{cn},\mathtt{superclasses}_{\mathtt{cn}}, \mathtt{structure}_{\mathtt{cn}}, \mathtt{instances}_{\mathtt{cn}}>$, where $\mathtt{cn} \in \mathtt{CN}$ is the class name, $\mathtt{superclasses}_{\mathtt{cn}}$ is the set of the names of the superclasses of `C`, $\mathtt{structure}_{\mathtt{cn}}$ is a set of pairs $\{<\mathtt{an}_1, \mathtt{x}_1>,.., <\mathtt{an}_{\mathtt{kcn}}, \mathtt{x}_{\mathtt{kcn}}>\}$ where for each pair $<\mathtt{an}_i, \mathtt{x}_i>$ $\mathtt{an}_i \in \mathtt{AN}$ and either $\mathtt{x}_i \in \mathtt{CN}$ or $\mathtt{x}_i \in \mathtt{DN}$. $\mathtt{Instances}_{\mathtt{cn}}$ is a set made of kcn sets of tuples $\{\mathtt{inst}_1,..,\mathtt{inst}_{\mathtt{kcn}}\}$ one for each pair $<\mathtt{an}_i, \mathtt{x}_i>$ belonging to $\mathtt{structure}_{\mathtt{cn}}$. Each tuple belonging to $\mathtt{inst}_j$ has the following structure: $<\mathtt{cn}: \mathtt{oid}, \mathtt{an}_j: \mathtt{y}>$, where $\mathtt{oid} \in \mathtt{OID}$ and `y` belongs either to `OID` or to `Dj` depending on the associate $\mathtt{x}_j$ is a class name or a domain name. We make the assumption that for all the kcn sets of tuples belonging to the instances of the class `C` the sets of `oids` corresponding to the tuple component labeled with `cn` coincide, and it is exactly the set of object identifiers of the class `C` itself. In other words, we postulate that each attribute of a class is characterized by at least one value. This is not a limitation, since we can easily split a class not satisfying such a property into a two level hierarchical structure, according to the following rules. The root class is characterized by the set of attributes that are always defined on all the instances of the original class; each of the child classes owns only one of the remaining attributes that, by definition, is always defined on the instances of the subclass itself. For instance, the class PERSON, having attributes NAME, AGE, CHILDREN, and OWNED_CARS, is represented by the hierarchy rooted at PERSON (with attributes NAME and AGE), having subclasses PERSON_WITH_SONS (with attribute CHILDREN) and PERSON_WITH_CAR (with attribute OWNED_CARS).

Let us define the mapping $\mathcal{H}$ between the object-oriented and the graph model: $\mathcal{H}(\mathtt{OODB}) = \mathrm{D}$, where D = <g, m, c> is a database expressed in terms of a typed graph g, an interpretation m, and a set of constraints c. The mapping $\mathcal{H}$ is specified as follows:

1) For each $\mathtt{dn}_i \in \mathtt{DN}$ and $\mathtt{D}_i = \mathtt{values}(\mathtt{dn}_i)$ there is a corresponding node $n_i \in N_{Cp}$ such that $f_1(n_i) = \mathtt{dn}_i$ and $m(n_i) = \mathtt{D}_i$;
2) For each $\mathtt{an}_i \in \mathtt{AN}$ there is a corresponding node $n_i \in N_R$ such that $f_1(n_i) = \mathtt{an}_i$ and $m(n_i)=\mathtt{inst}_i$, where $\mathtt{inst}_i$ is the set of tuples associated with the pair $<\mathtt{an}_i, \mathtt{x}_j>$, belonging to the structure of a class `cn`; moreover there exists a node $n_j \in N_C$ such that $f_1(n_j) = \mathtt{x}_j$ and there is the edge $<n_i,n_j>$;
3) For each $\mathtt{cn}_i \in \mathtt{CN}$ there is a corresponding node $n_i \in N_{Cn}$ such that $f_1(n_i) = \mathtt{cn}_i$ and its interpretation coincides with the set of object identifiers of the corresponding class; the adjacents of n_i, $AD(n_i)=\{n_1,...,n_{kcn_i}\}$, are such that: $\{f_1(n_1),...,f_1(n_{kcn_i})\}=\{a \mid <a,x> \in \mathtt{structure}_{\mathtt{cn}_i}\}$. Moreover, for each $y \in \mathtt{superclasses}_{\mathtt{cn}}$ it holds that : $f_1(n_i)$ ISA y.

Rule 1 states that the family of domains `D` corresponds to the set of interpretations of the printable nodes of g (i.e., N_{Cp}), and the names of such domains are given by the labels of the elements of N_{Cp}. Rule 2 states that the set `AN` of attribute names is equal to the set of labels of the role nodes (i.e., N_R) of g, and that the meaning of such nodes corresponds to the associated set of tuples in the `instances` of the classes the attributes belong to. Rule 3 states that the set `CN` of class names coincides with the set of labels of non-printable nodes in g, while the set of adjacent role nodes is derived by the `structure` of the class. The meaning of a non-printable node coincides with the set of the associated OIDs of the corresponding class.

In figure 4 an example of mapping between an object-oriented and a GMDBs is drawn. In fig.4a, through a very intuitive syntax, the object-oriented schema classes are described, while in fig.4b and 4c the corresponding graph model and interpretation are shown.

```
CLASS Vehicle    [
   Weight:Real;{<Vehicle:OI1,Real:200>,<Vehicle:OI2,Real:400>,<Vehicle:OI3,Real:150>};
   Manufacturer:Company;{<Vehicle:OI1,Company:OI4>,<Vehicle:OI2,Company:OI4>,
                         <Vehicle:OI3,Company:OI4>};
   ]
CLASS Automobile   [
   Superclasses:Vehicle;
   Plate:String(1:1);{<Automobile:OI1,String:CA400>,<Automobile:OI2,String:NY100>};
   Owner:Person(1:1);{<Automobile:OI1,Person:OI11>,<Automobile:OI2,Person:OI12>};
   ]
CLASS Company  [
   CompanyName:String(1:1);{<Company:OI4,String:GeneralMotors>,
                                <Company:OI5,String:Apple>};
   Location:String(1:1);{<Company:OI4,String:USA>,<Company:OI5,String:USA>};
   President:Employee(1:1);{<Company:OI4,Employee:OI7>,<Company:OI5,Employee:OI6>};
   Staff:Employee(1:n);{<Company:OI4,Employee:{OI8,OI9}>,
                         <Company:OI5,Employee:{OI10, OI11}>};
   ]
CLASS Person     [
   Name:String(1:1);{<Person:OI6,String:Steve>,<Person:OI7,String:Iacocca>,
                     <Person:OI8,String:Mark>,<Person:OI9,String:Kim>,
                     <Person:OI10,String:Peter>,<Person:OI11,String:Mary>,
                     <Person:OI12,String:Jules>};
   Age:Integer(1:1);{<Person:OI6,Integer:45>,<Person:OI7,Integer:49>,
                     <Person:OI8,Integer:22>,<Person:OI9,Integer:33>,
                     <Person:OI10,Integer:27>,<Person:OI11,Integer:37>};
   ]
CLASS Employee  [
   Superclasses:Person;
   Salary:Integer(1:1);{<Employee:OI6,Integer:60000>,<Employee:OI7,Integer:57000>,
                     <Employee:OI8,Integer:30000>,<Employee:OI9,Integer:25000>,
                     <Employee:OI10,Integer:29000>,<Employee:OI11,Integer:25000>};
   ]
```

Figure 4.a

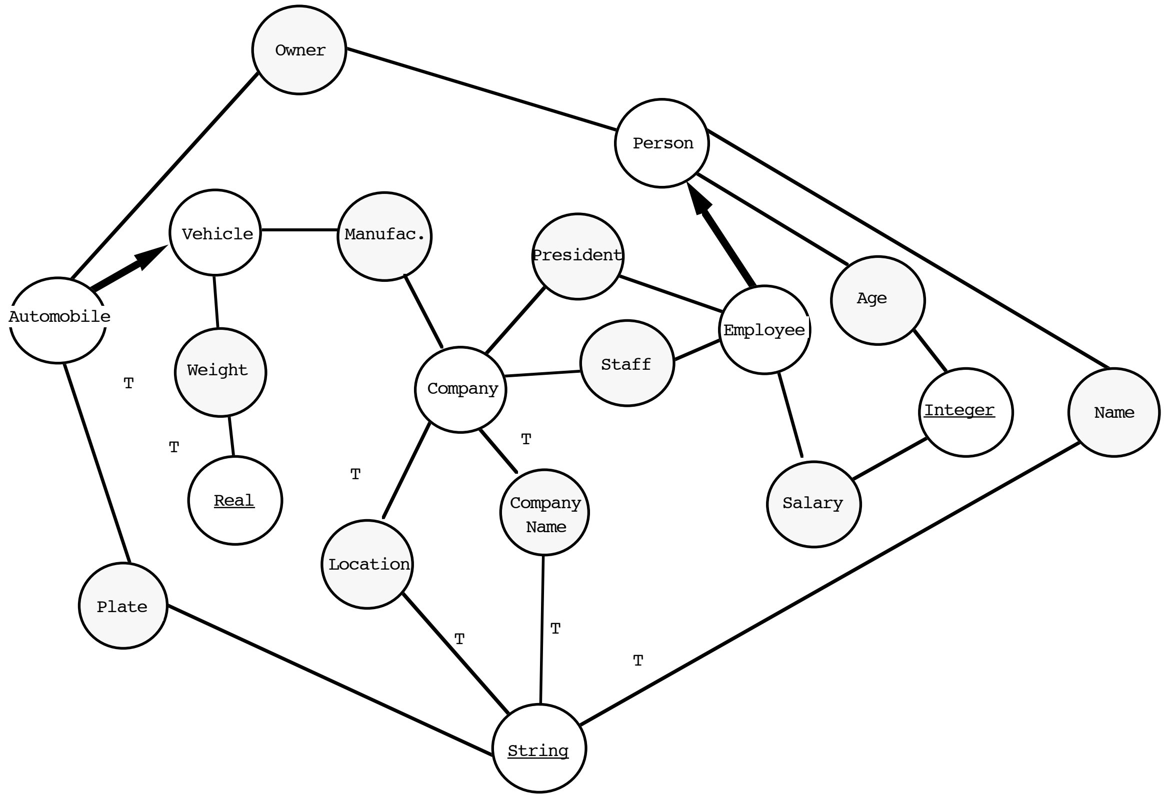


Figure 4.b

m (Person) = {OI6, OI7, OI8, OI9,OI10,OI11,OI12};
m (Vehicle) = {OI1,OI2,OI3};
m (Automobile) = {OI1, OI2};
m (Company) = {OI4, OI5};
m (Integer) = $\mathbb{Z}$;
m (Real) = $\mathbb{R}$;
m (String) = $\{a...z\}^*$;
m(Staff)={<Company:OI4,Employee:{OI8,OI9}>,<Company:OI5,Employee:{OI10, OI11}>};
m (Age) = {<OI6,45>,<OI7,49>,<OI8,22>,<OI9,33>,<OI10,27>,<OI11,37>};
m(Name)={<Person:OI6,String:Steve>,<Person:OI7,String:Iacocca>,
<Person:OI8,String:Mark>,<Person:OI9,String:Kim>,<Person:OI10,String:Peter>,
<Person:OI11,String:Mary>,<Person:OI12,String:Jules>};
m(President) = {<Company:OI4,Employee:OI7>,<Company:OI5,Employee:OI6>};
m(CompanyName)={<Company:OI4,String:GeneralMotors>,<Company:OI5,String:Apple>};
m (Plate) = {<Automobile:OI1,String:CA400>,<Automobile:OI2,String:NY100>};
m (Weight) = {<Vehicle:OI1,Real:200>,<Vehicle:OI2,Real:400>,<Vehicle:OI3,Real:150>};
m (Location) ={<Company:OI4,String:USA>,<Company:OI5,String:USA>};
m(Manufacturer)={<Vehicle:OI1,Company:OI4>,<Vehicle:OI2,Company:OI4>,
<Vehicle:OI3,Company:OI4>};
m(Owner) = {<Automobile:OI1,Person:OI11>,<Automobile:OI2,Person:OI12>};
m(Salary)={<Employee:OI6,Integer:60000>,<Employee:OI7,Integer:57000>,
<Employee:OI8,Integer:30000>,<Employee:OI9,Integer:25000>,
<Employee:OI10,Integer:29000>,<Employee:OI11,Integer:25000>};

Figure 4.c

Figure 4: An object-oriented database and the corresponding GMDB

Comparing the expressive power of the GPs against an object-oriented query language is not an easy task. In fact, whereas the relational model is provided with formally defined query languages (relational algebra and calculus), that constitute a well-known expressive power yardstick, the object oriented proposals lack a comprehensive model of queries. Indeed, several examples of different models of query language for object oriented databases are available in the literature [Banerjee et al. 1988; Kim Kim Dale 1989; Kim 1989]. Another complication arises from the structure of the result of a query on an object oriented database. In fact, the presence of class hierarchies leads to two equally valid interpretations for the access scope of a query involving a class c subsuming other classes:

- the scope is the set of objects satisfying the goals of the query and belonging to the class c, i.e., eventually belonging *also* to its subclasses;
- the scope is the set of objects satisfying the goals of the query and belonging *only* to c, i.e., not belonging to its subclasses.

In order to have a result characterized by a fixed number of attributes, we have to choose either the second of the above proposals or the first one, discarding the attributes of the subclasses of c. The latter is our approach.

In the following, in order to compare the expressive power of the GPs against an object oriented query model, we adopt the point of view of [Kim 1989]. Let us briefly recall the main features of such an approach. The queries are classified as single-operand and multiple-operand. The former, which are more strictly related to the object-oriented philosophy, consist in querying a single class, and are essentially based on the idea of extracting objects from a class by posing logical predicates on its attributes. The domain of the involved attributes determines two possible kinds of predicates: simple predicate, in the case of elementary domain, or complex predicate, if the attribute domain is a class. The structure of a simple predicate is <attribute-name, operator, value>, as usual. A complex predicate is a predicate on a contiguous sequence of attributes belonging to the structure of the class the attribute domain is based on. A single-operand query is more powerful than a single-relation query, because it implies joins of several classes when we make use of complex predicate. In fact, in this case a so-called implicit join is applied, which corresponds to the retrieval of objects from the target class according to pre-defined joins, stated by specifying another class as domain of an attribute of the target class. The basic limitation of the implicit join is that it statically determines the order in which the classes are to be joined. In other words, it is not possible to formulate a query whose semantics requires implicit reversal of an attribute-domain link specified between the classes.

Note that, if the structure of a class is cyclic, the single-operand query may express recursion. In the following we will not interact with such a case, since it is not immediate to graphically express the recursion. Moreover, also in textual languages, such as [Kim 1989], a suitable construct (e.g., "recurse") has to be added to a cyclic data structure in order to simulate a loop. In our case, we could graphically express recursive queries adding to the previously defined structures an icon representing the operation of looping. Notice that this icon is semantically overloaded, and its friendliness is questionable. On the other hand, it does not seem possible to express with purely graphical interactions complex operations, such as transitive closures, or more generally loops, and the iconic solution seems to be, at the present state of the art, the best for the final user (see [Catarci 1991]). However, since we are only interested in Graphical Primitives (not in iconic ones) we will not consider in the following such typologies of queries. A more detailed description of the different kind of cyclic data structure and of the associated recursive queries is in [Kim 1989].

In order to have a language whose expressive power is at least equivalent to the one of the relational algebra, single-operand queries are not sufficient, since joins of classes on user specified attributes (explicit join), and set-operations between different classes have to be admitted. At this aim, multiple operand queries are defined, i.e., explicit join and set-operations, extracting objects from two or more classes. Major problems arise in the case of set-operations, where the operands are sets of objects. The main difference between object-oriented and relational databases is that in the first case the operands may be heterogeneous sets of objects, while in the second are homogeneous sets of tuples. As a consequence, in the object-oriented model the result of such queries is a set of instances whose structure is not clearly defined. Various strategies have been proposed in order to overcome such a problem. In our opinion, set-operations would be allowed only between classes that are descendant of the same parent, having compatible types, since this seems to be the only meaningful case. On the other hand, set-operations between classes having fully different types and sets of objects may either be denied or give rise to classes having a structure resulting from the union of the two originating structures, thus with a large number of fields often containing null values. The introduction of such classes in the database could originate several problems in subsequent operations and introduce potential integrity violations. In the more simple case of explicit join, the result is a set of instances formed by concatenating the instances from the different joined classes.

In the following we introduce several examples in order to show how the graphical query primitives defined on the graph model allows us to express object-oriented queries. Note that we refer to the object-oriented database in figure 4.a and the corresponding one in the graph model in figures 4.b and 4.c. Let us consider the query "Find the names of all the USA companies whose president's age is greater than 46 years." This query falls in the class of single operand queries, as defined above, and it involves both a simple predicate (i.e., *Company.Location* = "USA") and a complex one (i.e., *Company.President.Age* > 46). The corresponding query in the language shown in [Kim 1989] is: select *Company* (*Location* = "USA" and *President Age* > 46). In our approach such a query is espressed by first selecting the nodes *Company, CompanyName, Location, President, Employee, Age, Integer*, and setting to displayed the nodes *Name* and *String;* then changing the labels of the edges <*Location*, *String*> and <*Age*, *Integer*> into *Location.String* = "*USA*" and *Age.Integer > 46* respectively. It is worth noting the advantage of handling in the same way both complex and simple predicates.

As a more intricate case, consider the query: "List the employees' names whose company president owns a car having the plate including the company name." The corresponding query, if reversible pointers are available, can be expressed in the language shown in [Kim 1989] as: select *Employee* (*Staff Name* IN *Staff President Owner Plate*); if the model does not includes such kind of pointers the query must be expressed as follows: select *Company* (*Name* IN *President Owner Plate*) and we can access the desired result (i.e., employees' names) as an attribute of Company. Note that while the semantics of the two queries is the same, the efficiency in evaluating them is largely improved in the first case. In principle, our approach does not suffer from this dichotomy, and the query is easily optimizable according to the well known techniques of query optimization (see, e.g., [Ullman 1987]). The above query is expressed by first selecting the nodes *Company, CompanyName, President, Employee, Owner, Plate, Staff, Automobile*, and setting to displayed the nodes *Name* and *String;* then drawing an edge between nodes *CompanyName* and *Plate* labeled with *CompanyName.String IN Plate.String*. Note that the drawing of an edge corresponds to the explicit join as mentioned above.

Summarizing, we have shown that we convert the structuring mechanisms of several data models into the ones of the graph model. We have presented only two examples, but we argue that the graph models may represent other data models as well. Moreover, it is worth noting that the graph model and the Graphical Primitives can be useful also for formally characterizing the visual representations of a generic data model and the semantics of graphical operations on it. In principle, that can be done for any triple <data model, representation, graphical query language>; on the other hand, the more is the semantic overloading associated with the graphical operations to characterize, the less is the naturalness of the approach. In the next section we will see an example of how the graph model and the Graphical Primitives are able to characterize a semantic model and the associated graphical query language.

6. Applications to Visual Query Systems

In this section we characterize the visual representation and the semantics of the graphical operations of the visual query language of SNAP [Bryce Hull 1986] which is representative of a large class of graphical systems based on semantic models. SNAP is based on the IFO model[Abiteboul Hull 1987; Hull King 1987] which is a mathematically defined database model, developed as a theoretical framework for studying the prominent semantic models.

In order to characterize the graphical query operations of SNAP we first need to define a mapping between the IFO data modeland the graph model. Then, since the syntax used in SNAP for representing IFO data schemas is defined in terms of different graphical elements (circles, diamonds, labels, edges, and connection rules), we also provide a mapping between the syntax of SNAP and the syntax of the graph model .

A *SNAP database* (SNDB) is a collection of object types, functional relationships, and ISA relationships. There are three kinds of atomic object types, and two constructs for recursively building more complex types. The three atomic types are called: 1) *printable*, which corresponds to objects of predefined types (e.g., integer, string, boolean, etc.); *abstract*, which corresponds typically to objects in the world that have no underlying structure (e.g., the type PERSON); a domain made of abstract objects is associated to this type; 3) *free*, which corresponds to entities obtained via ISA relationships. The names associated with the object types are called *tokens*. The first of the two mechanisms for constructing nonatomic types, called *association*, corresponds to the procedure of forming finite sets of objects of a given structure. The other mechanism is the well-known Cartesian product operator, called *aggregation* in the following. Functional relationships are represented by using *fragments*. Fragments are directed trees that may contain instances of all the different kinds of object types as vertices and labeled edges representing functions. The final structural component of the IFO modelare the ISA relationships. An ISA relationship from a type SUB to a type SUPER indicates that each object associated with SUB is associated with the type SUPER. This implies that each function defined on the type SUPER is automatically defined on the type SUB, i.e., it is inherited by SUB. Two types of ISA are distinguished: *specialization* and *generalization*. Specialization can be used for defining possible *roles* for members of a given type. In contrast, generalization represents situations where distinct, pre-existing types are combined to form new virtual types.

More formally, a *type* is a directed tree R = (V,E) such that: 1) V is the disjoint union of five sets: V_P (printable vertices), V_A (abstract vertices), V_{AS} (association vertices), V_{AG} (aggregation vertices), and V_F (free vertices); 2) printable, abstract, and free vertices are leaves of the tree; 3) association vertices have one child; 4) aggregation vertices have one or more children, which are viewed as being ordered. The set of *objects* of type R, denoted dom(R), is defined by:

1) if R is an atomic type, then dom(R) is a countably infinite set, made of atomic objects;
2) if the root r of R is an association vertex, and R_1 is the child subtree of R, then dom(R) = $\{\{O_1,...,O_m\} \mid m >= 0$, and for each i, O_i is in dom(R_1)$\}$;
3) if the root r of R is an aggregation vertex, and $R_1,...,R_n$ are the ordered child subtrees of r, then dom(R) = $\{<O_1,...,O_n> \mid$ for each i, O_i is in dom(R_i)$\}$.

An *instance* of R, denoted instances, is a finite subset of dom(R). A *fragment* is a directed tree R = (V,E) where:

1) E is the disjoint union of two sets E_O (*object edges*) and E_F (*functional edges*);
2) (V, E_O) is a forest of types;
3) the destination of each functional edge is the root of some type of R ;
4) the source of each functional edge is either the root of R or the child by an object edge of an association vertex, which is the root of some type of R , and not the root of R .

Note that rule 4 allows one to model nested functions. Let R be a fragment with root r, a nested function of degree n is represented by a path, named *functional path*, of the form: $<r,a_1,r_1,a_2,r_2,...,a_{n-1},r_{n-1},r_n>$, where a_i are association vertices and each r_i is the child by an object edge of the association vertex a_i. Let R be a fragment with root r; let $f_1 = <r,p_1>$, ..., $f_n = <r, p_n>$, be the functional edges of R with tail r; let R_0 be the type with root r; and for each k, k=1...n, let R_k be the maximal subtree of R with root p_k. An *instance* of R is an ordered pair I= (J,F) where J is an instance of R_O, and F a function with domain $\{f_1,..., f_n\}$ such that $F(f_k)$ is a partial function with domain J such that for each O in J: a) $F(f_k(O))$ is an object of R_k, if R_k is a type, and b) otherwise $F(f_k(O))$ is an instance of R_k', where R_k' is the fragment obtained from R_k

by removing its association root. A *database schema* is built starting from a forest of fragments, linked by ISA edges. An instance of a schema S is composed by an assignment of instances to all the fragments of S that satisfy certain conditions imposed by the ISA edges of S.

Let us now define the mapping $\mathcal{H}$ between the SNAP databases and the GMDBs: $\mathcal{H}$(SNDB) = D, where D = <g, m, c> is a database expressed in terms of a typed graph g, an interpretation m, and a set of constraints c. In order to specify the mapping $\mathcal{H}$, we make use of a function $\mathfrak{T}$, defined on the instances of a SNAP database. Let $x \in dom(R)$, if R is an atomic type, then $\mathfrak{T}(x) = x$; else $\mathfrak{T}(x) = new(x)$, where new is a function generating a new unprintable value when applied to a generic instance. The mapping $\mathcal{H}$ is specified as follows:

1) For each atomic type $v_i \in V_P$, there is acorresponding node $n_i \in N_{Cp}$ such that $f_1(n_i)$ is the name associated with v_i and $m(n_i)$ = instances(v_i);
2) For each atomic type $v_i \in V_A \cup V_F$, there is a corresponding node $n_i \in N_{Cn}$ such that $f_1(n_i)$ is the name associated with v_i and $m(n_i)$ = instances(v_i);
3) For each complex type R with root $v \in V_{AS}$ and child subtype R'(either complex or atomic), there are two nodes $n \in N_{Cn}$, $n_j \in N_R$ and an edge $<n,n_j>$, corresponding to v and such that $f_1(n)$ is the name associated with v, and a recursively defined tree corresponding to R',whose root n' is linked to n_j. The interpretation of n coincides with a set of new values generated by applying the new function to all the instances of v: m(n)={new(t), t $\in$ instances(v)}; the interpretation of n_j is a set of tuples of the form: $<f_1(n)$:new(y),$f_1(n')$:$\mathfrak{T}(x)>$, where y $\in$ instances(v) and x $\in$ instances(R').
4) For each complex type R with root $v \in V_{AG}$, with child subtypes $R_1,...,R_l$, there are a node $n \in N_{Cn}$, such that $f_1(n)$ is the name associated with v, a node $n_j \in N_R$ and the edge $<n,n_j>$ corresponding to the root v. Moreover, there are l recursively defined trees corresponding to the R_is whose roots n_is are linked to n_j. The interpretation of n coincides with a set of new values generated by applying the new function to all the instances of v: m(n)={new(t), t $\in$ instances(v)}; the interpretation of n_j is a set of tuples of the form: $<f_1(n)$:new(y),$f_1(n_1)$:$\mathfrak{T}(R_1)$,..$f_l(n_l)$:$\mathfrak{T}(R_l)>$, where y $\in$ instances(v) and $x_i \in$ instances(R_i), i=1...l.
5) For each functional edge e = $<r_0,x>$ with label l, where x is either the root of a type or an association node, there are: a node $n \in N_R$ with label l, one edge $<n_1,n>$, where n_1 is the root of the tree corresponding to the type with root r_0, and one edge $<n_2,n>$, where n_2 is either the root of the tree corresponding to the type with root x (if x is the root of a type) or is the unprintable node associated with the association node x (if x is an association node). The interpretation of n is a set of tuples of the form: $<f_1(n_1)$:$\mathfrak{T}(x_0)$,$f_1(n_2)$:$\mathfrak{T}(x_1)>$, where $<x_0,x_1> \in F(l)$.
6) For each ISA edge i = <a,b> there is a corresponding ISA constraint between n', which is the root of the tree corresponding to the atomic type with root a, and n", which is the root of the tree corresponding to the atomic type with root b.

Rule 1 (resp. rule 2) says that SNAP printable (resp. abstract and free) types are translated in terms of printable (resp. unprintable) class nodes. Rule 3 (resp. rule 4) says that each association (resp. aggregation) node generates a new unprintable class node, whose interpretation contains an identifier for each element belonging to the set of instances of the association node (resp. aggregations). Note that, given the recursive definition of IFO type instances, the definition of the node interpretations in rules 3 and 4 is recursive too. In rule 5 each functional edge is translated into a role node with two outcoming edges, whose interpretation implements the underlying instance function. Finally, ISA edges are rendered in ISA constraints.

In the following we define the semantics of SNAP graphical queries by expressing them in terms of GPs, whose semantics is always defined. As we said before, since the syntax of SNAP is defined in terms of various graphical elements (rectangles, circles, diamonds, labels, edges, and connection rules), we first need to provide a mapping $\mathcal{S}$ between SNAP diagrams (SNAPD) and Typed Graphs, .defined as follows:

1. for each rectangle $\mathtt{r} \in \mathtt{SNAPD}$ with label $\mathtt{lr}$, there is a node $n_i \in N_{Cp}$ such that $f_1(n_i) = \mathtt{lr}$;
2. for each diamond $\mathtt{rh} \in \mathtt{SNAPD}$ with label $\mathtt{lh}$, there is anode $n_i \in N_{Cn}$ such that $f_1(n_i) = \mathtt{lh}$;
3. for each circle $\mathtt{c} \in \mathtt{SNAPD}$ with label $\mathtt{lc}$, there is a node $n_i \in N_{Cn}$ such that $f_1(n_i) = \mathtt{lc}$;
4. for each star circle[1] $\mathtt{sc} \in \mathtt{SNAPD}$ with label $\mathtt{lsc}$, there is a pair of linked nodes, $n_i \in N_{C\,n}$ and $n_j \in N_R$, such that $f_1(n_i) = \mathtt{lsc}$;
5. for each cross circle[2] $\mathtt{pc} \in \mathtt{SNAPD}$ with label $\mathtt{lpc}$, there is a pair of linked nodes, $n_i \in N_{Cn}$ and $n_j \in N_R$, such that $f_1(n_i) = \mathtt{lpc}$;
6. for each unarrowed edge $\langle \mathtt{x}, \mathtt{y} \rangle \in \mathtt{SNAPD}$ (x is parent of y in a tree), there is an edge $\langle x', y' \rangle \in E$ such that $x' \in \mathcal{S}(\mathtt{x})$ [3] and $y' \in \mathcal{S}(\mathtt{y})$[4];
7. the set of arrowed edges belonging to a functional path $\mathtt{p}$ of lenght n, is translated into a role node z and n+1 edges of the form $\langle z, \mathcal{S}(\mathtt{r_i}) \rangle$, i=0..n, with $\mathtt{r_i}$ defined as above;

By using the mappings $\mathcal{H}$ and $\mathcal{S}$ we show that the query language of SNAP can be formalized in terms of several applications of GPs.

Queries in SNAP are expressed using query graphs, which are formed by combining one or more query fragments. Query fragments are constructed primarily from the fragments of a schema; a query graph can contain any number of query fragments, possibly including duplicates. The values that have to be associated with the query fragments are specified using mainly two mechanisms: *node restriction*, which permits users to associate a restriction directly with a given node of a fragment, *comparitor arcs*, which permit users to indicate that a certain relationship must hold between values associated with different nodes. The same comparitor arcs may also be used to compare abstract types, sets with sets, and individuals with sets. In such a way the label used on the comparitor arc is semantically overloaded. For instance, using an edge with label "inclusion" between two association nodes results in comparing the elements of the sets underlying the association nodes (which are sets themselves), verifying that the inclusion relation holds among such elements. Finally, the shading of nodes in a query graph indicates that only values associated with such shaded nodes have to be displayed in the answer.

Summarizing, the graphical operations available in SNAP are:
1) selection of a tree representing a fragment;
2) changing of a node label, which corresponds to node restriction;
3) shading of a node.
4) drawing of a labeled edge.

---

[1] Circles with an inscribed star are the representation for association nodes.

[2] Circles with an inscribed cross are the representation for aggregation nodes.

[3] $\mathcal{S}(\mathtt{x})$ produces a set of nodes and x' is the role node among them.

[4] If $\mathcal{S}(\mathtt{y})$ produces more than one node, y' is the class node.

Such operations, applied to a SNAP database SNDB, whose visual representation is a SNAP diagram SNAPD, correspond to several applications of GPs on the database D = <g, c, m> $=\mathfrak{H}$(SNDB) such that g = $\mathfrak{S}$(SNAPD). As a consequence, the semantics of such operations is immediately defined in terms of the semantics of the GPs.

The correspondence is as follows:

1) the selection of a tree f is expressed as: the selection of the nodes (denoted with $\mathfrak{n}(\mathfrak{S}(c_1))...\mathfrak{n}(\mathfrak{S}(c_k)))$ that are generated translating the tree components $c_1...c_k$, which are nodes, edges, functional paths: $\mathcal{A}(\mathfrak{H}(SNDB), \mathfrak{n}(\mathfrak{S}(c_1))) \circ ... \mathcal{A}(\mathfrak{H}(SNDB), \mathfrak{n}(\mathfrak{S}(c_k)))$;
2) the change of the label of a rectangle c with a new label $\mathfrak{F}$ is expressed as the change of the label of the edge linking $\mathfrak{S}$(c) to the (unique[1]) role node q with a new label $\mathfrak{F}'$, which is the corresponding of $\mathfrak{F}$ in the GP syntax: $\mathfrak{C}(\mathfrak{H}(SNDB), \mathfrak{F}', \mathfrak{S}(c), q)$;
3) the shading of a node p corresponds to setting to displayed the corresponding node in the GMDB: $\mathcal{A}(\mathfrak{H}(SNDB), \mathfrak{S}(p))$;
4) the drawing of a labeled edge (with label $\mathfrak{F}$) between two elements, l_1 and l_2, may correspond either to a single GP or to a sequence of applications of GPs, depending on the elements l_1 and l_2, namely:
   - if l_1 and l_2, are two rectangles, the corresponding GP is the drawing of an edge with a label $\mathfrak{F}'$, which is the corresponding of $\mathfrak{F}$ in the GP syntax, between the two (unique) role-nodes n_1 and n_2, linked with the printable nodes $\mathfrak{S}(l_1)$ and $\mathfrak{S}(l_2)$: $\mathfrak{E}(\mathfrak{H}(SNDB), \mathfrak{F}', n_1, n_2)$;
   - if either l_1 or l_2, or both, are star circles, the operation involves a comparison between sets, and cannot be directly expressed by a single GP. Instead, a sequence of GPs is necessary, which may be very complex, depending on the comparison operator specified on the edge. This is not surprising, since, as we said before, in this case the edge label is semantically overloaded, and the semantic distance between such operations and the GPs is big. For the sake of brevity, we do not give the details of this correspondence.

6. Conclusions

In this paper we have presented the graph model and the associated GPs. It is worth noting that the aim of the paper is not in defining a new data model. On the contrary, the basic idea is in giving importance to the visual representation, which is one of the most important components in building effective interfaces, and in defining a set of basic GPs, in terms of which more complex interactions may be expressed. In principle, the graph model can be used as the internal model of any visual query system, providing a sound formalism useful to characterize the semantics of the visual operations defined on the external model. Moreover, the GPs defined on the graph model can be effectively used for characterizing the semantics of the most popular visual query languages defined on both semantic and object-oriented data models.

Finally, since the generality of our approach allows the representation of a wide class of models, such as relational, semantic, and object-oriented, in terms of the graph model constructs, another application of the graph model concerns the problem of querying heterogeneous databases, i.e., databases characterized by the presence of different data models and different query languages. Our model can be seen as an "Esperanto Model," able to join different points of view resulting from the representation of several data models. In principle, for each model it is possible to define

[1] Note that the uniqueness of such a role node is a consequence of the IFO definition of fragment, not a graph model constraint.

a mapping to the graph model (examples are shown in [Catarci 1992]), thus providing a homogeneous representation for all data in the heterogeneous database.

Acknowledgments

We would like to thank Maurizio Lenzerini for many extremely helpful suggestions, ideas and comments.

References

[Abiteboul Hull 1987] S.Abiteboul, R. Hull - IFO: A Formal Semantic Database Model - *ACM Transactions on Database Systems, Vol. 12*, No. 4, pp. 525-565.

[Angelaccio Catarci Santucci 1990a] M.Angelaccio, T.Catarci, G.Santucci - QBD*: A Fully Visual Query System - *Journal on Visual Languages and Computing, Vol. 1*, No. 2, pp.255-273.

[Angelaccio Catarci Santucci 1990b] M.Angelaccio, T.Catarci, G.Santucci - QBD*: A Graphical Query Language with Recursion - *IEEE Transactions on Software Engineering, Vol.16*, No 10, pp.1150-1163.

[Bancilhon 1988] F.Bancilhon - object-oriented Database Systems - in Proc. ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Austin, Texas.

[Bancilhon Cluet Delobel 1990] F.Bancilhon, S.Cluet, C.Delobel - A Query Language for the O2 object-oriented Database System - in Proc. of the Second International Workshop on Database Programming Languages.

[Banerjee et al. 1988] J.Banerjee et al. - OQL: An object-oriented Query Language - in Proc. 4th Intl. Conf. on Data Engineering, Los Angeles.

[Batini et al. 1991] C.Batini, T.Catarci, M.F.Costabile, S.Levialdi - Visual Query Systems - Technical Report N.04.91 of Dipartimento di Informatica e Sistemistica, University of Rome "La Sapienza".

[Beeri 1990] C.Beeri - A Formal Approach to object-oriented Databases - *Data & Knowledge Engineering, 5*, pp.353-382.

[Bryce Hull 1986] D.Bryce, R.Hull - SNAP: A Graphics-based Schema Manager - Proc. of the IEEE Conference on Data Engineering, Los Angeles, USA, 151-164.

[Catarci 1991] T.Catarci - On the Expressive Power of Graphical Query Languages - in Proc. of the 2nd IFIP W.G. 2.6 Working Conference on Visual Databases, Budapest.

[Catarci 1992] T.Catarci - *Visual Languages for Querying Databases: Formalization and Expressive Power* - PhD Thesis, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza".

[Chang 1989] S.K.Chang - *Principles of Pictorial Information Systems Design* - Prentice-Hall, Englewood Cliffs, NJ.

[Chen 1976] P.P.Chen - The Entity-Relationship Model toward a Unified View of Data - in *ACM Transactions on Data Base Systems, Vol.1*, N.1.

[Codd 1970] E.F.Codd - A Relational Model for Large Shared Data Banks - *Communication ACM, Vol.13*, N.6, June.

[Codd 1972] E.F.Codd - Relational completeness of database sub-languages - In *Data Base Systems*, R.Rustin, Ed., Prentice Hall, Englewood Cliffs, 65-98.

[Consens Mendelzon 1990] M.Consens, A.O. Mendelzon - Graphlog: A Visual Formalism for Real Life Recursion - Proc. of the ACM Symp. on Principles of Database Systems, 404-416.

[Cruz 1990] I.Cruz - Declarative Query Languages for object-oriented Databases - Technical Report CSRI-238, Computer Systems Research Institute, university of Toronto, June.

[Cruz Mendelzon Wood 1988] I.F.Cruz, A.O.Mendelzon, P.T.Wood - G+: Recursive Queries Without Recursion - in Proc. of the 2nd International Conference on Expert Database Systems, April, pp. 355-368.

[Date 1987] C.J.Date - *An Introduction to Database Systems* - Vol.I, Addison-Wesley Publishing Company.

[Gyssens Paredaens, Van Gucht 1990] M.Gyssens, J.Paredaens, D. Van Gucht - A Graph-Oriented Object Model for Database End-User Interfaces - in Proc. ACM SIGMOD Conference on the Management of Data, Atlantic City, USA, pp. 24-33.

[Hull King 1987] R. Hull, R. King - Semantic Database Modeling: Survey, Applications and Research Issues - ACM Computing Surveys, Vol. 19, No. 3, September 1987, pp. 201 - 260.

[Hull King 1989] R.Hull, R.King - Semantic Database Modeling: Survey, Applications, and Research Issues - *ACM Computing Surveys, Vol.19*, N.3, 201-260.

[Kim 1989] W.Kim -A Model of Queries for object-oriented Databases - in Proc. of 15th Intl. Conf.on Very Large Database, Amsterdam.

[Kim 1990] W.Kim - *Introduction to object-oriented Databases* - Computer Systems Series, The MIT Press, USA.

[Kim Kim Dale 1989] K.C.Kim, W.Kim, A.Dale - Cyclic Query Processing in Object Oriented Databases - in Proc. 5th Intl. Conf. on Data Engineering, Los Angeles.

[Lenzerini Nobili 1990] M.Lenzerini, P.Nobili - On the Satisfiability of Dependency Constraints in Entyity-Relationship Schemata - *Information Systems, Vol. 15*, N.4.

[Levesque Mylopoulos 1979] H.Levesque, J.Mylopoulos - A Procedural Semantics for Semantic Networks - in N.V. Findler ed. *Associative Networks, Representation and Use of Knowledge by Computers*, Academic Press Inc.

[Mark 1989] L.Mark -A Graphical Query Language for the Binary Relationship Model. *Information Systems, 14,* 3, 231-246.

[Nanni 1988] U.Nanni - A Graphic Interface for Relational Databases. Technical Report, University of Roma, Italia.

[Shneiderman 1983] B.Shneiderman - Direct Manipulation: A Step beyond Programming Languages - *IEEE Computer, 16*, pp. 57-69.

[Ullman 1987] J.D.Ullman - *Principles of Database and Knowledge-Base Systems* - vol. I, Computer Science Press, Rockville, MD.