The VLDB Journal
© Springer-Verlag 1997

Graphical interaction with heterogeneous databases*

T. Catarci[1], G. Santucci[1], J. Cardiff[2]

[1] Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", Via Salaria, 113, I-00198 Rome, Italy
[catarci/santucci]@infokit.dis.uniroma1.it
[2] Department of Computing, RTC Tallaght, Dublin 24, Ireland; e-mail: jcar@staffmail.rtc-tallaght.ie

Edited by Y. Vassiliou. Received September 1993 / Accepted January 1996

Abstract. During the past few years our research efforts have been inspired by two different needs. On one hand, the number of non-expert users accessing databases is growing apace. On the other, information systems will no longer be characterized by a single centralized architecture, but rather by several heterogeneous component systems.

In order to address such needs we have designed a new query system with both user-oriented and multidatabase features. The system's main components are an adaptive visual interface, providing the user with different and interchangeable interaction modalities, and a "translation layer", which creates and offers to the user the illusion of a single homogeneous schema out of several heterogeneous components. Both components are founded on a common ground, i.e. a formally defined and semantically rich data model, the Graph Model, and a minimal set of Graphical Primitives, in terms of which general query operations may be visually expressed. The Graph Model has a visual syntax, so that graphical operations can be applied on its components without unnecessary mappings, and an object-based semantics.

The aim of this paper is twofold. We first present an overall view of the system architecture and then give a comprehensive description of the lower part of the system itself. In particular, we show how schemata expressed in different data models can be translated in terms of Graph Model, possibly by exploiting reverse engineering techniques. Moreover, we show how mappings can be established between well-known query languages and the Graphical Primitives. Finally, we describe in detail how queries expressed by using the Graphical Primitives can be translated in terms of relational expressions so to be processed by actual DBMSs.

1 Introduction

Recently, the database area has been proven 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]. *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 functions 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 recent 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.

Various graphical VQSs have been proposed (a survey is in [Batini et al. 1991]), but few of them are provided with a formal definition (e.g., [Cruz Mendelzon Wood 1988; Nanni 1988; Angelaccio Catarci Santucci 1990; Consens 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. VQSs have been shown to be appropriate for querying a global information system [Batini et al. 1991], because they typically offer to the user a representation of the information that is independent of the structure or the location of the actual data. Nevertheless, existing VQSs generally do not adapt to the various needs of different users. Neither do they interface heterogeneous databases, a critical need of today information systems (see, e.g., [Thomas et al. 1990; Sheth and Larson 1990; Elmagarmid Pu 1990]).

On the contrary, the main goal of our approach is to allow different classes of users to access multiple, heterogeneous databases by means of an adaptive interface, offering several interaction mechanisms. This led us to design and

* Research partially supported by the EC under the Esprit Project 6398 VENUS.

partially implement a query system with both user-oriented and multidatabase features. One of the basic ideas of the system is to give a precise semantics, in terms of changes of database states, to a set of elementary graphical actions (such as selection of nodes and drawing of edges), called *Graphical Primitives* (GPs), in terms of which more complex visual interaction mechanisms may be precisely defined. The GPs are defined on the basis of a powerful data model, the *Graph Model* (GM), having a visual syntax and an object-based semantics.

On one hand, the GM is powerful enough to express the semantics of most of the common data models, so it is suitable as a unifying canonical model. A user can query and examine results using a conceptually single database, namely a *Graph Model Database* (GMDB). Then, her/his query is translated into a set of queries which are executed on the component databases, and the results of these are combined to form a single result. The user is thus oblivious to the existence of the underlying databases, and need not be concerned with their specific storage formats or query languages. On the other hand, the GPs can be used as basic constituents of more complex visual interaction mechanisms and different visual representations can be associated with the GMDBs by simple syntactic mappings.

The work on the above query system has evolved during several years, producing various partial results. This paper is a comprehensive description of the lower part of the global system. More precisely, we show how schemata expressed in different data models can be translated in terms of GMDBs, possibly by exploiting reverse engineering techniques. Moreover, we show how mappings can be established between well-known query languages and the GPs, and vice versa. Finally, we describe how queries expressed by using the GPs can be transformed into relational expression so to be processed by actual database management systems (DBMSs; we concentrate on relational DBMSs, since they are widely diffused in real applications). Other aspects of this research are described elsewhere. In particular, the multiparadigmatic user interface is presented in [Catarci Chang Santucci 1994] and the GM foundations are in [Catarci Santucci Angelaccio 1993].

This paper is structured as follows: In Sect. 2 we describe the system architecture. The basic notions on the GM are recalled in Sect. 3. Section 4 gives the translations between the GM and the relational, object-oriented, and semantic data models. The query management and the result construction are described in Sect. 5. Finally, the conclusions are presented in Sect. 6, and the directions in which this work will proceed are summarized.

2 The overall system architecture

According to the main goals described in Sect. 1, we propose a global system with the following basic features:

– A graph-based formalism (namely the GM) for representing and querying databases. This formalism is suitable to give a precise semantics to complex visual representations and is general enough to formalize, in principle, a database expressed in any of the most common data models. The querying primitives of the formalism, although constituted solely by two elementary graphical actions, namely the selection of a node and the drawing of an edge, are at least as expressive as the relational algebra.
– An adaptive visual interface, built on the basis of the above formalism, providing the user with different visual representations and interaction mechanisms together with the possibility of switching among them. All the different visual representations allow one to express at least the class of conjunctive queries.
– The definition of three suitable sets of translation algorithms, one for translating a database expressed in any of the most common data models into the internal system model, one for translating a GM query in terms of the query languages of the underlying data models and one devoted to implement the consistent switching among different visual representations during the query formulation.
– The construction and the management of an effective user model, which allows the system to provide the user with the most appropriate visual representation according to his or her skill and needs.

The architecture of the whole system is shown in Fig. 1. The system consists of a *Visual Interface Manager*, a *User Model Manager*, a *GMDB & Query Manager*, and one or more DBMSs.

The Visual Interface Manager is capable of supporting multiple representations (form-based, iconic, diagrammatic, and hybrid) of the databases and the corresponding interaction modalities. In this way the user is provided with a Multiparadigmatic Query Language that is based on a set of visual query languages, each of them interacting with a different visual representation of the GM and all sharing the same expressive power. We briefly recall the features of the representations the Visual Interfac e Manager interacts with.

Form-based representations are the first attempt to provide the user with friendly interfaces for data manipulation; they are usually proposed in the framework of the relational model, where forms are actually tables. Their main characteristic consists in visualizing prototypical forms where queries are formulated by filling corresponding fields. In the precursor systems adopting a form-based representation, like QBE [Zloof 1977], only the intensional part of relations is shown: the extensional part is filled by the user in order to provide an example of the requested result. In more recent proposals both the intensional and the extensional part of the database coexist.

Diagrammatic representations are those most used in existing systems. Typically, they represent with different visual elements the various types of concepts available in a model. The correspondence between visual elements and related types of concepts demands aesthetic criteria for the placement of visual elements and connections. For example, hierarchical structures for generalization and object aggregation dictate a vertical placement of the involved elements. Diagrammatic representations adopt as typical query operators the selection of elements, the traversal on adjacent elements and the creation of a bridge among disconnected elements.

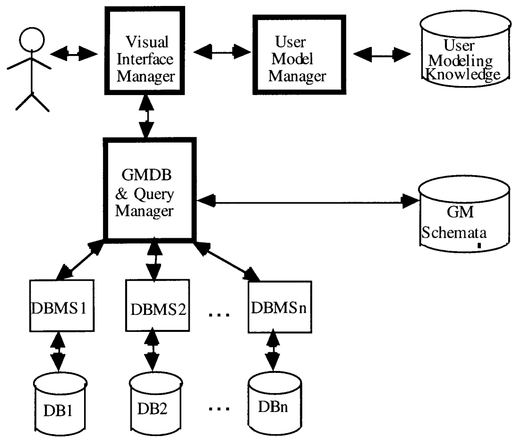


Fig. 1. The system architecture

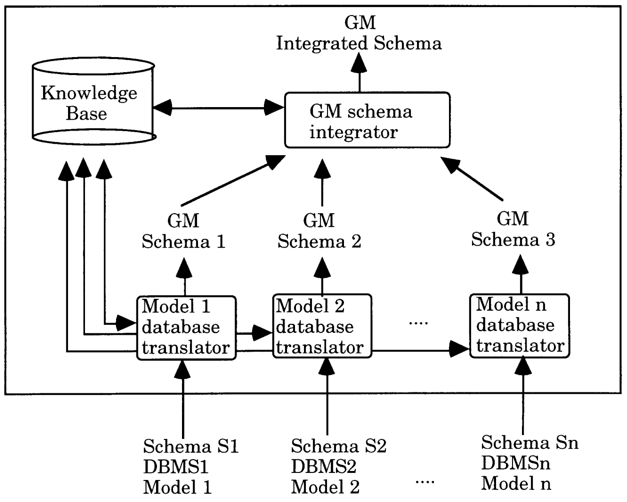


Fig. 2. Conversion and integration of local schemata

The iconic representation uses sets of icons to denote both the objects of the database and the operations to be performed on them. A query is expressed primarily by combining icons. For example, icons may be vertically combined to denote conjunction (logical AND) and horizontally combined to denote disjunction (logical OR) [Chang 1990]. In order to be effective, a proposed set of icons should be easily understandable by most people. However, in many cases it is difficult or even impossible to find a universally accepted set of icons. As an alternative, icons could be user-defined to be tailored to the particular needs of the user and to her/his own mental representation of the tasks s/he wants to perform.

The *hybrid representations* use an arbitrary combination of the above approaches, either offering to the user various alternative representations of databases and queries, or combining different visual structures into a single representation. From an analysis of VQSs [Batini et al. 1991], it emerges that several systems adopt more than one visual structure, but often one of them is predominant. On the contrary, in the proper hybrid VQSs, the different visual structures share the same significance. Diagrams are often used to describe the database schema, while icons are used either to represent specific prototypical objects or to indicate actions to be performed. Forms are mainly used for displaying the query result.

Based upon the user model provided by the User Model Manager, the Visual Interface Manager selects the visual representation most appropriate for the user. The User Model Manager is responsible for collecting data and maintaining a knowledge base of the user model components, namely the class stereotype, the user signature, and the system model. The Visual Interface Manager and the User Model Manager are described in more detail in [Catarci Chang Santucci 1994] and [Catarci et al. 1993].

At the bottom of the figure, different databases structured according to several data models are shown. Each database is translated into a GMDB by the GMDB & Query Manager, using the mappings described in Sect. 4. It is up to the GMDB & Query Manager to manage such mappings and to translate the visual queries into queries that can be executed by the appropriate DBMS.

Figures 2 and 3 give more details about the GMDB & Query Manager, showing its main components. Figure 2 describes the initial activity of the GMDB & Query Manager, i.e., the conversion of the DBMS schemata into GMDB schemata and the following integration.

Each of the merging schemata is expressed in terms of a data model supported by a suitable DBMS. The GMDB & Query Manager activates the needed translators to convert the local schemata in terms of GMDB schemata. During this phase, each translator interacts with an internal knowledge base for two main objectives:

1. To document the choices adopted during the translation activity
2. To find out additional information (if available) about the schema it is translating

Once the set of GMDB schemata is available, the GM schema integrator module provides for integrating them. The first step in the integration of two schemata is to recognize their similarities; these provide the starting point for the integration. However, the principal difficulty of schema integration is to discover and solve possible conflicts in the schemata to be merged, i.e., different representations for the same concepts.

Conflict analysis aims at detecting all the differences in representing the same reality in the schemata. The main research guidelines (see, for instance, [Navathe Gadgil 1982; Batini Lenzerini 1984; Gottard Lockemann Neufeld 1992] for specific methodologies and [Batini Lenzerini Navathe 1986] for a comparative survey), indicate several kinds of schema differences detectable by conflict analysis, including:

1. *Naming conflicts*: Schemata incorporate names for entities, attributes, and relationships. People from different application areas of the same organization are used to refer to the same data using their own different terminology and names. This results in a proliferation of names as well as a possible inconsistency among names in the component schemata. The problematic relationships among names are of two types:
   – Homonyms: when the same name describes two different concepts, giving rise to inconsistency unless detected.

  - Synonyms: when the same concept is described by two or more names, giving rise to a proliferation of names.
2. *Domain conflicts*: Insidious inconsistencies may exist in the definition of the domain of concepts having the same name in two different schemata. For instance, the entity *Student* can include graduate and undergraduate students in one of the two schemata, and undergraduate students only in the other one.
3. *Structural conflicts*: These are conflicts that arise because of a different choice of modeling constructs or integrity constraints. Examples of conflicts are:
   - Type inconsistencies: The same concept is represented by different modeling constructs in different schemata (e.g., the use of city as an entity in one schema and as an attribute in another one).
   - Cardinality ratio conflicts: a group of concepts are related among themselves with different cardinality ratios in different schemata (e.g., Man and Woman in the relationship 'Marriage' are 1:1 in one schema, but m:n in another one, accounting for a marriage history).
   - Key conflicts: different keys are assigned to the same concept in different schemata.

Because of the limited set of structural mechanisms of the GM, the conflicts which arise are almost always name conflicts, and the GM schema integrator keeps track of the (eventual) renaming of concepts, documenting it in the internal knowledge base. Moreover, the knowledge base will contain also pieces of information about the global schema restructuring. This activity is performed by further analyzing the global schema against the main goals of completeness and minimality. Eventually, the internal knowledge base documents the distribution of the data among the merging schemata as well.

In Fig. 3 we show the modules of the GMDB & Query Manager devoted to the query management.

Through the Visual Interface Manager the user interacts with a view of the integrated GM schema and her/his actions are translated in terms of GPs. Once the query is completed, the user can ask for its computation. The Query Handler module, through the analysis of the distribution information produces a set of admissible views on the local GM schemata. Each view is processed by the appropriate query translator, resulting in a query expressed in terms of the underlying DBMS. Each DBMS computes its partial query and sends the result to the Query Handler that, in turn, merge them, producing a unique result.

3 Graph Model and Graphical Primitives

In this section we recall from [Catarci Santucci Angelaccio 1993] the formal definition of the concepts underlying our approach. We first introduce the syntax and the semantics of the Graph Model in terms of *Typed Graph* and *Interpretation*, and then we define a suitable language for expressing *Constraints* on the elements of the Typed Graph. Afterwards, we describe the GPs that allow for expressing any query-oriented user interaction with a database in terms

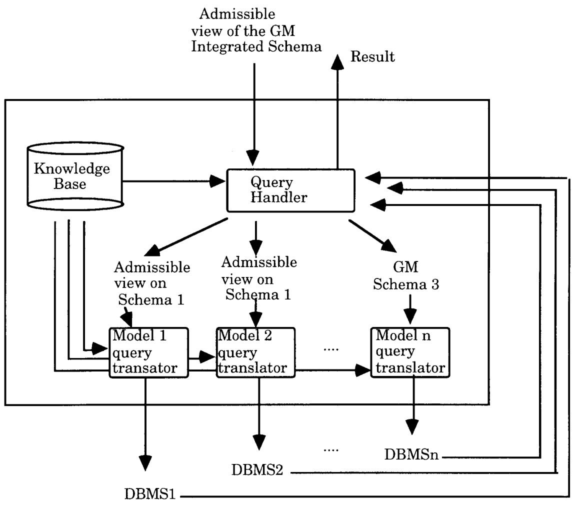


Fig. 3. The Query Management

of two simple graphical operations: the selection of a node and the drawing of a labeled edge.

3.1 The Graph Model

The GM allows us to define a GMDB D in terms of a triple $\langle g, c, m \rangle$, where g is a *Typed Graph*, c is a set (possibly empty) of integrity *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.

The intensional part of the database is expressed in the Typed Graph in terms of classes and relations (called roles). A class is an abstraction of a set of objects with common characteristics, whereas a relation among classes $C_1, \ldots, C_n$ represents associations among objects of the classes $C_1, \ldots C_n$. More formally, we define the Typed Graph as follows.

Definition (Typed Graph)
A Typed Graph g is a 7-tuple: $g = \langle N, E, \mathscr{L}_1, \mathscr{L}_2, f_1, f_2, f_3 \rangle$ where:

- $N = N_C \cup N_R$ is the set of nodes, where N_C and N_R are mutually disjoint; 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_u}, the set of unprintable class-nodes.
- $E \subseteq N \times N$ is the set of edges.
- $\mathscr{L}_1$ is the set of node labels.
- $\mathscr{L}_2$ is the set of edge labels, including a special label T^1.
- f_1 is a total one-to-one function from N to $\mathscr{L}_1$.
- f_2 is a total function from E to $\mathscr{L}_2$.
- f_3 is a total function mapping each node to one value in $\{\texttt{unselected}, \texttt{selected}, \texttt{displayed}\}$.

[1] All edges are labeled T at the beginning of the interaction. Note that such a label is for the purpose of the system and it is not displayed

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

The second component of a GMDB, namely the integrity constraints, allows the designer to specify relevant conditions on and meaningful properties of the classes and the roles represented by the nodes in the Typed Graph. It is worth noting that the user simply aiming at querying the database does not need to be acquainted with the existence of constraints. Instead, constraints are essential in specifying the database schema, since they represent an important segment of the semantics of the application. In this paper we are interested in two kinds of integrity constraints: the ISA constraints, and the cardinality constraints.

The ISA constraints allows for representing subclass-class relationships. More precisely, we can impose an ISA constraint on a Typed Graph g by simply enforcing that one class-node C of g is ISA-related to another class-node D of g. As we will see in the definition of interpretation, this means that C represents a sub-class of D in g.

Cardinality constraints allow for limiting the number of participations of objects in relations. They come in two forms, the ATLEAST and ATMOST forms. The first form is written as ATLEAST(k, C, R), and is used to assert that every object that is an instance of the class-node C is linked to at least k instances of the role-node R. The second form is written ATMOST(k, C, R), and is used to assert that every object that is an instance of the class-node C is linked to at most k instances of the role-node R.

The ISA, ATLEAST and ATMOST constructs are graphically represented in a Typed Graph as shown in Fig. 4a, i.e., an arrowhead edge for the ISA relationships, and a pair of numbers between brackets for the cardinality constructs.

Let us now turn our attention to the third component of a GMDB, namely, the Interpretation. In defining the notion of Interpretation, we use the following notations:

– $AD\{n_1, \ldots, n_k\}$ is the set of nodes adjacent to a given set of nodes $\{n_1, \ldots, n_k\}$ *minus* $\{n_1, \ldots, n_k\}$.
– $\mathscr{D} = \mathscr{D}_p \cup \mathscr{D}_u$ is a set of elementary objects. $\mathscr{D}_p$ is a set of printable objects, $\mathscr{D}_u$ is a set of unprintable objects. Moreover, it holds that $\mathscr{D}_p \cap \mathscr{D}_u = \emptyset$.
– $\mathscr{U}$ is a universe for a Typed Graph g, that is a set of structured objects, defined as the smallest set containing $\mathscr{D}$ and all the possible labeled tuples (of any arity) over $\mathscr{D}$, i.e. objects of the form $\langle l_1 : t1, \ldots, l_n : tn\rangle$, where $l_1, \ldots, l_n$ and $t_1, \ldots, t_n$ are elements of $\mathscr{L}_1$ and $\mathscr{D}$, respectively.

Given a universe $\mathscr{U}$, an Interpretation for a Typed Graph g over $\mathscr{U}$ (or simply an Interpretation for g) is a function mapping the printable class-nodes of g to a subset of all printable objects of $\mathscr{U}$, the unprintable class-nodes to a subset of all unprintable objects of $\mathscr{U}$, and the role-nodes to a subset of all labeled tuples of $\mathscr{U}$. In particular, given a role-node n, its Interpretation is constituted by a set of labeled tuples whose arity is equal to the number of class-nodes which are adjacent to n, and each component is labeled with the label of one adjacent class-node, and takes its values in the corresponding Interpretation. Formally, an Interpretation is defined as follows.

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

– If $n \in N_{C_p}$ then $m(n) \subseteq \mathscr{D}_p$.
– If $n \in N_{C_u}$ then $m(n) \subseteq \mathscr{D}_u$.
– If $n \in N_R$ and $\{n_1, \ldots, n_h\} = AD\{n\} \cap N_C$, then $m(n)$ is a set of tuples of the form $\langle f_1(n_1) : t_1, \ldots, f_1(n_h) : t_h\rangle$, where $f_1(n_i) \in \mathscr{L}_1$ and $t_i \in m(n_i)$ for $i = 1..h$.

Moreover, the interpretation is said to satisfy a set of constraints c if the following conditions are satisfied:

– For every constraint $C$ISA$D$ in c (i.e., for every constraint enforcing that C is a subclass of D), we have that $m(C) \subseteq m(D)$.
– For every constraint ATLEAST(k, C, R) in c (i.e., for every ATLEAST cardinality constraint on C and R), the number of labeled tuples in $m(R)$ that contain elements of $m(C)$ in the C-component is greater than or equal to k.
– For every constraint ATMOST(k, C, R) in c (i.e., for every ATMOST cardinality constraints on C and R), the number of labeled tuples in $m(R)$ that contain elements of $m(C)$ in the C-component is less than or equal to k.

In the following, when we refer to a GMDB $D = \langle g, c, m\rangle$, we implicitly assume that the interpretation m satisfies every constraint in c. We note that, given a Typed Graph g and a set of constraints c, there always exists at least one Interpretation 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.

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

– If $n \in N_C$ then $AD'\{n\} = AD\{n\} \cup_i AD(n_i)$ where $n_i \in N_{C_u}$ and nISA$^* n_i$ holds, where ISA* is the transitive closure of the ISA relation (note that if $n \in N_{C_p}$, then $AD'\{n\} \neq AD\{n\}$).
– If $n \in N_R$ then $AD'\{n\} = AD\{n\} \cup \{m \in N_C | \text{n} \in AD'\{m\}\}$.

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 the use of the GM for the specification of a database concerning persons, students, and cities, is shown in Fig. 4. Figure 4a shows the Typed Graph g with the following constraints: *Student* ISA *Person*; ATLEAST$(1, Person, Age)$; ATMOST$(1, Person, Age)$; ATLEAST$(1, Person, Lives)$; ATMOST$(1, Person, Lives)$, i.e., a person has one and only one age and lives in one and only one city. Figure 4b shows a possible Interpretation m for g.

Observe that the values stored in the unprintable nodes play the role of object-identifier and they can be suitably changed without affecting the information content of the schema. For instance, if we create a new GMDB, say D_2, from the one in the above example, changing all occurrences of the object identifier *OI1*, belonging to the class *Person*, into *OI100*, we can assert that the two GMDBs are equivalent. Therefore we say that two GMDBs, say D_1 and D_2, belong to the same equivalence class if they share the same Typed Graph and the Interpretation of D_2 is obtained by the Interpretation of D_1 by applying an isomorphic function to all the unprintable values of the Interpretation of D_1. In the following we denote the equivalence class of a GMDB D_1 as $EQ(D_1)$.

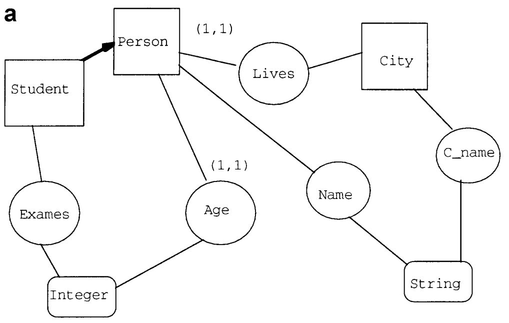
We end this section with a discussion on the main features of the GM. The formal definition of the GM shows that our formalism shares many features with other well-known data models proposed in the literature. In particular, the class-oriented nature of the model allows us to stress on one hand several similarities with semantic modeling, and on the other hand many important differences with record-oriented data models (relational, hierarchical and network data models). The intensional part of the database is indeed expressed in the Typed Graph in terms of classes and relations, in the tradition of conceptual semantic data models [Hull King 1987], as well as frame-based knowledge representation formalisms [Brachman Schmolze 1985]. The kind of integrity constraints supported by the model is also tailored to the class-oriented nature. ISA constraints are used to represent the ISA relation, which is one of the most relevant semantic relations on classes. The importance of the ISA relation in representing knowledge is stressed in many papers (see, for example, [Brachman 1993]), and stems from the fact that it allows a modular approach to schema specification, based on the inheritance of properties. On the other hand, cardinality constraints are important in establishing dependencies along classes. For example, it is easy to see that suitable forms of functional and existence dependencies between classes are indeed captured by ATMOST and ATLEAST constraints, respectively.

Although there are several similarities between the GM and the Semantic Data Models, we can also single out some distinguishing features of our formalism. We will discuss this point by referring to three important class-oriented data models.

The GM can be considered a variant of the extended Entity-Relationship Model [Chen 1976; Ullman 1987], in that class-nodes can be interpreted as entity types and role-nodes as relationship types. Unlike the Entity-Relationship Model, our formalism does not distinguish between attributes and relationships in specifying properties of a class. Indeed, we think that this distinction is not meaningful from the user point of view. Instead, the important point for the user is to have an effective and direct means to single out logical links among classes (or entities), and this is accomplished in the GM by links relating class-nodes and role-nodes.

There are several similarities between the GM and the Functional Data Model [Shipman 1981]. Indeed, the Functional Data Model has an easily understood visual representation, and is again based on the representation of classes and links. However, representing complex associations among classes is not an easy task when using the Functional Data Model, whereas the GM provides the notion of role-nodes

a



b

m (Student) = {OI1, OI2, OI3};
m (Person) = {OI1, OI2, OI3, OI4};
m (City) = {OI5, OI6};
m (Integer) = Z;
m (String) = $\{a...z, 1...0\}^*$;
m (Age) = {<Person:OI1,Integer:22>, <Person:OI2,Integer:21>, <Person:OI3,Integer:19>,
<Person:OI4,Integer:40>};
m (Name) = {<Person:OI1,String:Mary>, <Person:OI2,String:John>,
<Person:OI3,String:Carl>, <Person:OI4,String:Cecily>};
m (Exams) = {<Student:OI1,Integer:5>, <Student:OI2,Integer:4>, <Student:OI3,Integer:1>};
m (C_name) = {<City:OI1,String:NY>, <City:OI6,String:Boston>};
m (Lives) ={<Person:OI1,City:OI5>,<Person:OI2,City:OI5>,<Person:OI3,City:OI6>,
<Person:OI4,City:OI6>};

Fig. 4a,b. A Graph Model DataBase. **a** A Typed Graph g and a set of Constraints c. **b** An example of Interpretation for $\langle g, c\rangle$

for this purpose. We remind the reader that a role-node is interpreted as a labeled tuple, and this directly reflects the fact that its instances are associations among objects that are instances of classes. Similar considerations hold in comparing the GM with the Binary Data Model [Abrial 1974].

Finally, it is useful to briefly compare the GM to Object-Oriented Database Models [Bancilhon 1988; Kim 1990; Beeri 1990]. It is easy to see that the Typed Graph inherits two characteristics of the object-oriented database approach. First, a distinction is made between abstract and concrete (also called printable) classes. The former represent sets of objects denoted simply by object identifiers, whereas the latter represent sets of objects that are actually values of distinguished domains (integers, reals, characters, etc.). Second, the ISA relation is treated similarly in the two formalisms, and the (possibly multiple) inheritance of properties is a central notion of both models. Unlike the Object-Oriented Data Models, though, the GM does not support a strong notion of complex object. Indeed, record and set structures are not explicitly modeled in our formalisms, but they are represented implicitly in role-nodes and in ATMOST cardinality constraints. The choice of not modeling complex objects in Typed Graph is mainly motivated by the fact that users are typically not familiar with record and set structures, that are instead much more oriented to computer science experts.

All the above observations point out that, although the GM embeds many characteristics of well-known data models, its graphical and class-oriented nature represents a somewhat novel approach to data modeling, that is especially suited to our goal of stressing both the user-oriented and the multidatabase features of the presented query system.

3.2 Fundamental Graphical Primitives

In this section we recall the formal definition of the 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 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.

The function $DUPLICATE^k(D)$, where $D = \langle g, c, m\rangle$ is a GMDB, results in a new GMDB $D^k = \langle g^k, c^k, m^k\rangle$ (the k-copy of D) which is equal to D except for the node labels ($\circ$ denotes concatenation of two labels); in particular:
$N_k = N$.
$E_k = E$.
$\mathscr{L}_1^k = \{k \circ l | l \in \mathscr{L}_1\}$.
$\mathscr{L}_2^k = \mathscr{L}_2$.
$f_1^k = \{\langle n, k \circ f_1(n)\rangle\}$.
$f_2^k = f_2$.
$f_3^k = f_3$.
$c_k = c$.
m^k is equal to m except for the labels of the tuple components of the role-nodes:

$$m^k(n) = \{\langle k \circ l_1 : t_1, \ldots, k \circ l_k : t_k\rangle \\ |\langle l_1 : t1, \ldots, l_k : t_k\rangle \in m(n)\} .$$

3.2.1 Selection of a node and drawing of an edge

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

Selection of a node n in D : $\mathscr{S}(D, n)$
$\mathscr{S}(D, n) = D'$ 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'.

Drawing of a labeled edge in D : $\mathscr{E}(D, \mathscr{F}, n, q)$
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 $\mathscr{F}$, which may be a boolean expression, the symbol "$\equiv$", or a set-oriented operation.

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

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 Sect. 3.2.2) this will give rise to a restriction of the final Interpretation.

Let n and q be unprintable class-nodes. The database $D' = \mathscr{E}(D, \text{“} \equiv \text{”}, n, q)$ is such that:
$N' = N$.
$E' = E \cup \{\langle n, q\rangle\}$.
$\mathscr{L}_1' = \mathscr{L}_1$.
$\mathscr{L}_2' = \mathscr{L}_2 \cup \{\equiv\}$.
$f_1' = f_1$.
$f_2' = f_2 \cup \{\langle\langle n, q\rangle, \equiv\rangle\}$.
$f_3' = f_3$.
$c' = c$.
m' is equal to m except for $m'(x)$, where $x \in AD(q)$:

$$m'(x) = \{\langle l_1 : v_1, \ldots, l_k : v_k, f_1(n) : v_{k+1}\rangle | \langle l_1 : v_1, \ldots, \\ l_k : v_k, f_1(q) : v_{k+1}\rangle \in m(x)\} .$$

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 $\mathscr{F}$ 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, $D' = \mathscr{E}(D, \mathscr{F}, n, q)$ is such that:
$N_C' = N_C \cup \{s\}$ (s is a new class-node).
$E' = E$.
$\mathscr{L}_1' = \mathscr{L}_1 \cup \{et_s\}$, where et_s is a new label for s.
$\mathscr{L}_2' = \mathscr{L}_2$.
$f_1' = f_1 \cup \{\langle s, et_s\rangle\}$.
$f_2' = f_2$.
$f_3' = f_3 \cup \langle s, \texttt{displayed}\rangle$.
c' is equal to c plus new ISA constraints concerning n, q and s, namely:
If $\mathscr{F} = \cup$ then $c' = c \cup \{n\text{ISA}s, q\text{ISA}s\}$.
If $\mathscr{F} = \cap$ then $c' = c \cup \{s\text{ISA}n, s\text{ISA}q\}$.
If $\mathscr{F} = -$ then $c' = c \cup \{s\text{ISA}n\}$.
m' is equal to m except for $m'(s)$: $m'(s) = m(n)\ setop\ m(q)$.

The above 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 $\mathscr{C}(D, \mathscr{F}, s, q)$, where $\mathscr{F}$ 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 Sect. 3.2.2) 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.

3.2.2 Result database

We have seen in the previous subsection that the result of applying a GP to a GMDB 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 performed on D. 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°, taking into account the restrictions specified in the labels of the edges drawn during the query phase.

More formally, 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 $m \in N_{C_p}$ and $f_3(m) =$ `displayed`. Let $m_1, \ldots, m_h \in N_{C_p}$ 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:

$N_R^\circ = \{new(r_1), .., new(r_k)\}$.
$N_{C_u}^\circ = \{q\}$.
$N_{C_p}^\circ = \{new(m_1), \ldots, new(m_h)\}$.
$E^\circ = \{\langle new(m_i), new(r_j)\rangle | \langle m_i, r_j\rangle \in E\} \cup \{\langle q, new(r_j)\rangle | r_j \in \{r_1, \ldots, r_k\}\}$.
$\mathscr{L}_1^\circ = \{et_q\} \cup \{f_1(r_i) | r_i \in \{r_1, \ldots, r_k\}\} \cup \{f_1(m_i) | m_i \in \{m_1, \ldots, m_h\}\}$.
$\mathscr{L}_2^\circ = \{T\}$.
$f_1^\circ = \{\langle q, et_q\rangle\} \cup \{\langle new(x), et\rangle | \langle x, et\rangle \in f_1$ and $x \in \{r_1, \ldots, r_k, m_1, \ldots, m_h\}\}$.
$f_2^\circ = \{\langle e, T\rangle | e \in E^\circ\}$.
$f_3^\circ = \{\langle x,$ `unselected`$\rangle | x \in N^\circ\}$.
$c^\circ = \emptyset$.

In order to specify m°, i.e., the Interpretation of $q, r_1, \ldots, r_k$, 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$f_3(x) \in \{$`selected`, `displayed`$\}$ and there exists no $w \in N_C$ such that $x\text{ISA}^*w$ and $w \in \{$`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: $TAD(y) = \{x | x \in AD'(y)$ 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.

$$\begin{aligned} RM(m(y)) = \{\langle l'_1 : v_1, \ldots, l'_k : v_k\rangle | &\langle l_1 : v_1, \ldots, l_k : v_k\rangle \\ &\in m(y) \text{ and } l'_i : v_i \text{ are such that} \\ &l'_i = f_1(TAD(y) \cap \{z \in N_C | z\text{ISA}^* f_1^{-1}(l_i)\} \text{ and} \\ &v_i \in m(f_1^{-1}(l'_i)))\,. \end{aligned}$$

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 ($\circ$ denotes concatenation of two labels):

$$\begin{aligned} RM'(m(y)) = \{\langle l''_1 : v_1, \ldots, l''_k : v_k\rangle | &\langle l'_1 : v_1, \ldots, l'_k : v_k\rangle \\ &\in RM(m(y)) \text{ and } l''_i = f_1(y) \circ l'_i\}\,. \end{aligned}$$

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

$m^\circ(q) = \{t_1, \ldots, t_s\}$, 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 $RIS = \emptyset$.
- If $|N_R^\circ| = 1$ and $|\{k \in N_R | f_3(k) \in \{$`selected`, `displayed`$\}\}| = 1$ then $RIS = RM'(m(k))$, where $k \in N_R$ and $f_3(k) \in \{$`selected`, `displayed`$\}$.
- Otherwise, $N_R^\circ = \{r_1, \ldots, r_k\}$, with $k \geq 1$ and $RIS = inst(eval(n_1, eval(n_2, \ldots, eval(n_{h-1}, n_h))))$, where $\{n_1, \ldots, n_h\} \equiv \{m \in N_R | f_3(m) \in \{$`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:

$$\begin{aligned} &\{\langle l_1 : v_1, \ldots, l_k : v_k, \text{"}nc\text{"} \circ l_{k+1} : v_{k+1}, \ldots, \\ &\text{"}nc\text{"} \circ l_h : v_h, l_{h+1} : v_{h+1}, \ldots, l_t : v_t\rangle\} \\ &\text{such that for } i = k+1 \ldots h, f_1^{-1}(l_i) \in N_{C_u} \text{ and} \\ &\langle l_1 : v_1, \ldots, l_k : v_k, f_1(n_1) \circ l_{k+1} : v_{k+1}, \ldots, \\ &f_1(n_1) \circ l_h : v_h\rangle \in RM'(m(n_1)) \\ &\langle f_1(n_2) \circ l_{k+1} : v_{k+1}, \ldots, f_1(n_2) \circ l_h : v_h, \\ &l_{h+1} : v_{h+1}, \ldots, l_t : v_t\rangle \in RM'(m(n_2))\,. \end{aligned}$$

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 $\mathscr{F}_1, \ldots, \mathscr{F}_n$ 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 $\langle l_1 : v_1, \ldots, l_y : v_y, f_1(q) : v_{y+1}\rangle$ such that $v_{y+1} \in m^\circ(q)$, $\langle l_1 : v_1, \ldots, l_y : v_y\rangle \in RIS$ and satisfies $\mathscr{F}_1, \ldots, \mathscr{F}_n$, 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°:

$$\begin{aligned} m^\circ(r_i) = \{\langle l : v, f_1(q) : v_{y+1}\rangle | &\langle l_1 : v_1, \ldots, f_1(r_i) \circ l : v, \ldots, \\ &l_y : v_y, f_1(q) : v_{y+1}\rangle \in RIS'\}\,. \end{aligned}$$

3.3 Example

As we said in the introduction, the aim of the GPs is mainly to provide a new approach to the VQLs formalization. We think that such primitives have to be essentially used as a formal basis for building more powerful and friendly visual query operators, as we do in the Multiparadigmatic Query Language. Nevertheless, the GPs may be directly used for querying the GMDBs, as shown in the following example.

Referring to the GMDB in Fig. 4, we are interested in retrieving the names of all the students whose age is greater than 21 and living in NY. We assume that the GMDB D is such that:

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

Assuming that the GPs are directly available to the user, the sequence of operations s/he has to perform in order to express the above query is the following:

– Switch to `selected` the state of the nodes *Student*, *Age*, *Lives*, *City*, *C_name*, and *Integer*; switch to `displayed` the state of the nodes *Name* and *String*. These actions produce the schema view in Fig. 5a.
– Change the label of the edge $\langle C\_name, String\rangle$ into $C\_name.String$ ="NY"; change the label of the edge $\langle Age, Integer\rangle$ into $Age.Integer > 21$. The resulting Typed Graph is shown in Fig. 5b.

The system evaluates the result database and displays the corresponding GMDB D° (see Fig. 5c). The only individual satisfying the conditions specified in the above phases is *OI1*, having *Name* "Mary", *Age* 22, and *C_name* "NY". A new object, namely *OI20*, is created for D°. It constitutes the interpretation of the node *RESULT*, with value "Mary" for the relationship *Name*.

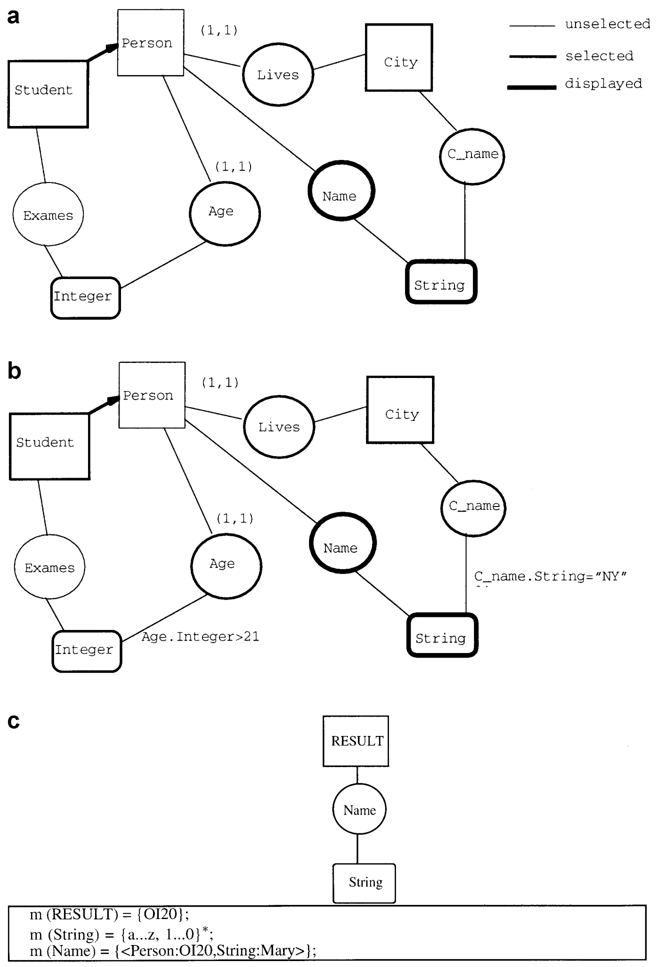


Fig. 5a–c. An example of query formulation

4 Database translation

Translating between models has been the subject of much research since the mid-1970s. The work done in this area can be divided into two classes, depending on the relative expressiveness of the models. The problem has been effectively solved when the translation is to a less expressive model, with well-defined transformations being defined to map constructs in one model to constructs in the other. For example, transformation algorithms to convert an Entity-Relationship schema to relational, hierarchical, and network schemata are given in [Elmasri Navathe 1989].

The second class of translation is from a semantically weaker to a semantically richer model. This is a much harder task as it involves acquiring additional semantics about the schema, apart from that which is expressed in the model. Reverse engineering can be loosely defined as the process of enriching a schema defined on a model with semantics not expressible in this model, in order to translate it into an equivalent schema defined on a more expressive model. Although reverse engineering could be applied between any pair of models, the literature typically concentrates on transforming relational schemata into more expressive schemata. This is due to a combination of the prevalence of relational databases, and to its weak expressive power compared to most other data models. Relational schemata tend to be the most difficult to reverse engineer, and translation of other models is usually comparatively trivial. For these reasons, we also concentrate on the problems of adding semantics to schemata defined on the relational model.

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 GM constructs. In this section we show how to map databases expressed in several data models, namely relational, semantic, object-oriented, into GMDBs. Our final goal is to produce an integrated schema of several heterogeneous databases. In order to do this, the first step is to apply reverse engineering techniques to semantically weaker data models, such as the relational one, so as to obtain, as far as possible, semantic equivalence between the different data models. Thus, we propose, in Sect. 4.2, a methodology for obtaining a semantically enriched GMDB starting from a relational database and a set of constraints. Sections 4.3 and 4.4 define the mappings between object-oriented and semantic databases and GMDBs respectively.

4.1 Relational model

The relational model [Codd 1970] is widely used in existing DBMSs, since it is formal and offers a simple and uniform data structure. 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 $DN = \{dn_1, .., dn_n\}$ be a set of *domain names*, and let $D = \{D_1, .., D_n\}$ be a family of *domains*, each composed by a set of *values* (we suppose to have finite domains); let $eval$ be a function, $eval : DN \rightarrow D$, associating a domain to each name; let $A = \{a_1, \ldots, a_m\}$ be the *set of attribute names*, and let $dom : A \rightarrow D$ be a function which associates a domain $dom(a)$ to each $a \in A$. Let $RN = \{rn_1, \ldots, rn_k\}$ be a set of *relation names*, and let $attr : RN \rightarrow 2^A$ be a function which associates to each $rn \in RN$ a non empty set $attr(rn)$ of attributes, called *relation schema*, and let $inst$ be a function, which associates to each $rn \in RN$ a finite set $inst(rn)$, called *relation instance* and composed of tuples of the form $\langle a_1 : v_1, a_2 : v_2, \ldots, a_h : v_h\rangle$ where $\{a_1, \ldots, a_h\} = attr(rn), v_i \in dom(a_i)$. A relation is a triple $r = \langle rn, attr(rn), inst(rn)\rangle$, with $rn \in RN$. A *relational database* $DB = \{r_1, \ldots, r_k\}$ is a set of relations with different names.

In [Catarci Santucci Angelaccio 1993] we introduced the concept of *relational GM database*, which is a GMDB constrained to obey the relational model rules. In particular, each role-node is forced to represent a function linking an unprintable class-node to a printable class-node. This reflects the structure of a relational schema, in which the unprintable 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. In the same paper, the mapping from relational databases to GMDBs as well as the reverse one were introduced.

Roughly speaking, a correspondence is established between: (1) the family of domains D and the set of interpretations of the printable nodes of g (i.e., N_{C_p}); (2) the set A of attribute names and the set of labels of the role-nodes (i.e., N_R) of g; (3) the attribute domains and the interpretation of the printable nodes which are adjacent to the role-nodes (note that each role-node has a single adjacent printable node as said above); (4) the set RN of relation names and the set of labels of unprintable nodes in g; (5) for each rn, the set of its attribute names $attr(rn)$ and the set of labels of the role-nodes which are adjacent to the corresponding unprintable class-node; (6) the instance set of each relation rn and the set of tuples obtained by joining the interpretations of the role-nodes sharing the same tuple component coming from the interpretation of the unprintable node corresponding to rn; (7) each relation r and a triple whose first element is an unprintable 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 Fig. 6 an example of mapping between relational databases and GMDBs is drawn.

4.2 Relational model with constraints

One of the main criticisms of the relational model has been its lack of expressiveness – a particular weakness is its power to model entities. Rather than representing entities in a unified structure, this information is spread across several tables, with the only connection between the tables being foreign key linkages. When a relational database schema is constructed, much of the information about what the entities are, how they are categorized, and how they relate to each other is lost in the transformation. This “semantic gap” must be bridged in some way – in the case of a user, this may be by knowledge in her/his head of the meaning of the relations, or in the case of an application, by semantics actually coded into the programs.

Why would it be necessary to reverse engineer a schema? There are two possible scenarios. Firstly, if the system is being migrated to a more expressive physical data model, it will be necessary to translate the relational schema to one containing more semantics. Secondly, if the system is to become part of a heterogeneous DBMS, then the relational schema must take part in a federated schema, which will be built using a canonical data model, typically based on the object-oriented paradigm.

The missing semantics must in both cases be provided from some source. In many cases the paper-based semantic data model which was used to construct the schema initially will have been lost, or modifications will have been made which are not reflected in the initial design. In this case, reverse engineering must be applied. There is no formal or well-accepted method to perform this task – the designer must glean information from a variety of sources, which include:

- Conversations with analysts familiar with the domain.
- The application programs accessing the database.
- The integrity constraints. These may be explicitly provided, or they may be inferred from an analysis of application programs or user dialogue.
- Data mining techniques.
- Analyzing query patterns etc.

The process of reverse engineering is more of an art than a science, and often some of the decisions made may be incorrect. Clearly, the more information that is available, the greater the likelihood that a correct schema will be arrived at, but in any case, the process is an iterative one, which will require constant feedback and verification from domain experts.

4.2.1 Overview of reverse engineering techniques

Reverse engineering can be seen as a subset of the area of schema translation. This has received much consideration in the literature. [Tsichritzis Lochovsky 1982] presents the various issues involved in translating between data models. This approach, along with most others, concentrates of defining mappings between specific pairs of model. In [Kalman 1989], an approach is given which allows translations between any two data models, based on a denotational semantics of data model equivalence.

a

PERSON

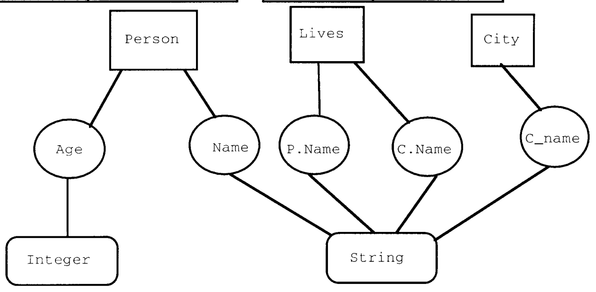
Age	Name
22	Mary
21	John
19	Carl
40	Cecily

LIVES

P.Name	C.Name
Mary	NY
John	NY
Carl	Boston
Cecily	Boston

CITY

C_name
NY
Boston



b

```
m (Person) = {OI1, OI2, OI3, OI4};
m (City) = {OI5, OI6};
m (Lives) = {OI7, OI8,OI9,OI10};
m (Integer) = Z;
m (String) = {a...z}*;
m (Age) = {<Person:OI1,Integer:22>, <Person:OI2,Integer:21>, <Person:OI3,Integer:19>,
    <Person:OI4,Integer:40>};
m (Name) = {<Person:OI1,String:Mary>, <Person:OI2,String:John>,
           <Person:OI3,String:Carl>, <Person:OI4,String:Cecily>};
m (P.Name) = {<Lives:OI7,String:Mary>, <Lives:OI8,String:John>,
               <Lives:OI9,String:Carl>, <Lives:OI10,String:Cecily>};
m (C_name) = {<City:OI5,String:NY>, <City:OI6,String:Boston>};
m (C.Name) = {<Lives:OI7,String:NY>, <Lives:OI8,String:NY>,
              <Lives:OI9,String:Boston>,<Lives:OI10,String:Boston>};
```

Fig. 6a,b. A relational database and the corresponding GMDB

Neither of these approaches focus how to translate between models of less semantic expressiveness to models of greater expressiveness, which is the problem being addressed in reverse engineering. An algorithm is given by [Navathe et al. 1987] to convert a relational schema into an Entity Category relationship, using keys to find the relationships between relations. This work is extended in [Johanssen et al. 1989] to eliminate certain anomalies and to consider inclusion dependencies, with the output being an Extended Entity-Relationship (EER) model. In [Davis Arora 1987], a methodology is given to translate between a relational model and an entity-relationship model. Keys and name equalities are used to obtain the structure of the model. Some weaknesses have been identified with this approach [Yan 1992], specifically entity fragmentation caused by multivalued dependencies, and assumptions are made about dangling keys which may be incorrect. In [Briand et al. 1987], a minimum cover of Functional Dependencies, Multivalued Dependencies, and Join Dependencies are used to produce an EER schema.

Translation of relational schemata to object-oriented schemata is considered by [Castellanos Saltor 1991] and [Yan 1992]. The former is motivated by the need for semantic enrichment to facilitate interoperability, and the constraints used are inclusion dependencies and several other categories of dependency. The class definitions created by this approach are rather complicated, and may result in inefficient query translations. An informal approach to translate from a relational to an object-oriented schema is presented in [Yan 1992]. Key and inclusion constraints, along with user input is used to extract the object-oriented semantics.

In the following section, we describe a methodology for reverse engineering we will apply to relational databases in order to obtain GMDBs. This methodology follows a sequence of formal steps in order to obtain a semantically enriched GMDB starting from a relational database plus a set of integrity constraints.

4.2.2 The mapping to the Graph Model

In this section, we consider how to represent an enriched relational database as a GMDB. We first consider the steps we must take in the process of semantically enriching the relational schema. We must be able to identify

1. What are the main entities.
2. What are the relationships between these entities.
3. What are the multivalued attributes of the entities (these need to be stored separately in the relational model).
4. What are the subclass/superclass relationships that hold between the entities.

In order to answer these questions, it is necessary to have knowledge of the integrity constraints that hold between the attributes. The minimal set required is the complete set of referential integrity constraints (foreign keys), although it will also be useful to have available inclusion dependencies, equality constraints, exclusion constraints, and domain constraints.

Here, we expand the definition of the relational model given in Sect. 4.1 to include the concepts of primary key and foreign key. The *primary key* of a relation $r \in DB$ is defined as follows. Let K be a subset of $attr(rn)$ such that for any value of $inst(rn)$, the projection of r on K has the same number of tuples as $inst(rn)$ and this property does not hold for any proper subset of K. The primary key PK_r of r is arbitrarily chosen among the set of possible K's. A relation r contains a *foreign key* $FK_{r,s}$, of a relation s, if the projection of r on $FK_{r,s}$ is contained in the projection of r on PK_r. If r and s coincide then $FK_{r,s}$ must not be the primary key.

In order to represent the relational model with constraints in terms of the GM, 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.

If the relational schema is enriched, the constraints on the links between the nodes given in Sect. 4.1 no longer apply. Links can be defined between unprintable class-nodes in order to express relationships explicitly. Role-nodes need not have a maximum of two edges, in order to allow relationships themselves to have attributes. Also the constraint $\mathrm{ATMOST}(1, n, q)$ is removed so that multivalued attributes can be represented.

In order to build a GMDB schema from a relational database schema DB, we follow the steps outlined above. We note that in general there are two important categories of relation – "entity relations", which informally equate to the backbone of the entity, and "relationship relations", which describe relationships between the entities. We refer to these sets of relations as DB_E and DB_R respectively. Also we say that an attribute $r.A$ is inclusion dependent on attribute $s.B$ (r and s not necessarily distinct) if there is an inclusion dependency $r.A \supseteq s.B$ defined.

We denote with $\mathscr{H}$ the mapping from a relational database to the corresponding database expressed in terms of the GM. Furthermore, we denote with $RelExp(n)$ the relation resulting from applying the relational expression associated with the node n when defining its meaning. We observe that $RelExp(n)$ involves only one relation, say r, and always contains the primary key of r, so it follows that $PK_{RelExp(n)} \equiv PK_r$. Moreover, if $RelExp(n)$ contains a foreign key of a relation s, it holds that $FK_{RelExp(n),s} \equiv FK_{r,s}$.

If DB is any relational database, $\mathscr{H}(DB) = D = \langle g, c, m\rangle$ is defined by the following steps.

Step 1: Identify entity relations
We can identify the main entity relations by adopting the following rule: a relation r is a member of the set of entity relations DB_E if there is no subset of the primary key which is a foreign key or is inclusion dependent on any other attribute in any other relation. For each $r \in DB_E$, we define $T_r(\Pi_{PK_r}(r))$ as an isomorphic function which returns a new value in D. The inverse function Tr^{-1} is defined as well. Note that T_r works also on foreign keys of r, i.e., it works on any $FK_{x,r}$. Finally, for each entity relation r, we define FK_r as the union of all the foreign keys $FK_{r,x}$ in r (if any).

Action: For each entity relation r, we perform the following actions.

1. For each $dn \in DN$ and $D = values(dn)$, there is a corresponding node $n \in N_{C_p}$ such that $f_1(n) = dn_i$ and $m(n) = D$.
2. For each relation $r \in DB_E$ there is a corresponding node $n \in N_{C_u}$ such that $f_1(n) = r$ and its interpretation coincides with the set of values generated by applying the T_r function to the tuples belonging to $inst(r) : m(n) = \{T_r(a) | a \in \Pi_{PK_r}(r)\}$; moreover, for each $a \in attr(rn)$, whose corresponding node is m (see 3), there is the edge $\langle n, m\rangle$ in E.
3. For each $a \in attr(rn)$—FK_r there is a corresponding node $n \in N_R$ such that $f_1(n) = a$ and there is an edge $\langle n, m\rangle$, where $m \in N_{C_p}$ is the node corresponding to $dom(a)$. Moreover, $m(n) = \{\langle a : x, b : y\rangle | \langle T_r^{-1}(x), y\rangle \in \Pi_{PK_r \cup a}(r)\}$.

Step 2: Identify relationships
Relationships between entities are represented in relational schemata by two means, depending on the cardinality constraints of the relationship.

Case 1: Relationship is 1:1 or 1:n

In the case of one-to-one or one-to-many relationships, a relationship is represented through a foreign key embedded in an entity relation. For the former, the foreign key can exist in either of the relations, but in the latter, it must be placed in the relation on the "many" side of the relationship.

Action: Let r and s be two entity relations such that a foreign key in r is the primary key of s, i.e. there exists $FK_{r,s}$. Let n and $m \in N_{C_u}$ be two nodes corresponding to relations r and s respectively. Define a new role-node, $u \in N_R$. Let $f_1(u)$ be a label descriptive of the relationship being defined by the foreign key. Let $E = E \cup \{\langle n, u\rangle, \langle m, u\rangle\}$. The following constraint holds: $\text{ATMOST}(1, u, n)$.

The meaning $m(n) = \{\langle a : x, b : y\rangle | \langle T_r^{-1}(x), T_s^{-1}(y)\rangle \in \Pi_{PK_r \cup FK_{r,s}}(r)\}$.

Case 2: Many-to-many relationships

Many-to-many relationships are represented as relationship relations. If a relation r represents a relationship, we can partition its primary key PK_r into $FK_{r,s_1}, FK_{r,s_2}, \ldots, Fk_{r,s_n}$, where $s_1 \ldots, s_n$ are previously defined entity relations. If $attr(rn)$—PK_r is not null, it represents attributes of the relationship.

Action: Let $n_1, n_2 \ldots, n_n \in N_{C_u}$, correspond to the entity relations $s_1 \ldots, s_n$ respectively. Define a new role-node, n, and let $E = E \cup \{\langle n_1, n\rangle, \langle n_2, n\rangle, \ldots \langle n_n, n\rangle\}$. If $attr(rn)$—$PK_r \neq \emptyset$, then assume it is composed of attributes $a_1 \ldots a_l$ and there is a corresponding set of nodes $m_1 \ldots m_l \subseteq N_{C_p}$ such that $m(m_i) = dom(ai)$. Let $E = E \cup \{\langle n_r, n_p\rangle\}$, for each n_p. The meaning $m(n) = \{\langle f_1(n_1) : x_1, \ldots, f_1(n_n) : x_n, f_1(m_1) : v_1, \ldots, f_1(m_l) : v_l\rangle | \langle T_{s_1}^{-1}(x_1), T_{s_n}^{-1}(xn), v_1, \ldots, v_l\rangle \in inst(rn)\}$.

Step 3: Include multivalued attributes
Multivalued attributes need to be stored in individual relations. These relations can be recognised by the form $r_i(K)$, where K, the primary key spans the entire relation, and is of the form (PK, A), where PK is the primary key of another relation, say r_j, (the entity relation), and A is the multivalued attribute. There will also be either an inclusion dependency or an equality dependency between this and the entity relation.

Action: Let $n_p \in N_{C_p}$ be a node such that $m(n_p) = dom(A)$, and n_r a new role-node. Let $E = E \cup \{\langle n_p, n_r\rangle, \langle n_c, n_r\rangle\}$, where n_c is the unprintable class-node corresponding to the relation r_j, with the cardinality constraint $\text{ATMOST}(m, n_c, n_r)$, where $m > 1$. The meaning $m(n_r) = \{\langle f_1(n_c) : x_1, f_1(n_p) : x_2\rangle | \langle T_{r_j}^{-1}(x_1), x_2\rangle \in inst(ri)\}$.

Step 4: Construct class hierarchy
There are two alternative approaches to representing subclasses in relational database schemata. These represent a trade-off between space usage and performance. A subclass can either be defined in a separate relation to its superclass, or alternatively the attributes pertaining specifically to a subclass can be defined as part of the schema of the entity relation. The first approach is more efficient disk usage, but performance will suffer as one or more joins will be required to obtain all information pertaining to the entity. The second approach results in a lot of null values, but queries can be executed more efficiently. Also, the semantics of the schema are clearer if a separate relation is used. We will consider these cases separately, although both the representations may be used in one relational schema.

Case 1: Subclass defined in a separate relation

Let r_1 and r_2 be two relations, such that r_1 is an entity relation and r_2 represents a subclass of this entity. In this case, there is an inclusion dependency $K_1 \supseteq K_2$, where K_1 and K_2 are the primary keys of relations r_1 and r_2.

Action. Define a node $n \in N_{C_u}$, corresponding to the relation r_2. Follow through steps 1 to 3 for this relation, in order to identify any associated relationships and multivalued attributes. Let $n_1 \in N_{C_u}$ be the node associated with the relation r_1. Define the following link: $n_2 \text{ISA} n_1$.

Case 2: Subclass embedded in entity relation

In order to identify the class hierarchy where subclasses are embedded in the entity relations, it is not sufficient to consider the relational schema and the constraints. Instead, we must look for clues in the extension of the relation. If sensible results are to be obtained from this process, we should consider a "significant sample", i.e., it should be of adequate size to make valid inferences on the data, and it should be representative of typical data.

We propose the following algorithm to identify the subclasses embedded in a relation $r(A_1 \ldots A_n)$.

1. Let $t_1 \ldots t_p$ be a significant sample of tuples in r. Partition these tuples into sets $T_1 \ldots T_q$ where each $t_i \in T_j$ has the same pattern of null/non-null values.
2. Build a matrix M with the horizontal axis corresponding to the attributes of r, and the vertical axis corresponding to the T_is just formed. The entries in the matrix are defined as follows: if the attribute A_i in group T_j is not null, $M[i,j]$ = 1, else $M[i,j]$ = 0.
3. Partition the attributes according to groups $G_1 \ldots G_m$ such that for any $A \in G_i$, it has the same pattern of entries $M[i,j]$, $1 \leq j \leq n$.
4. For each pair of groups G_i and G_j, show an ISA link between G_j and G_i if for each attribute $A_j \in G_j$ supports a superset of the null values supported attributes $A_i \in G_i$.
5. Eliminate transitive ISA links. The result is a class hierarchy.

As this is a heuristical approach, it is necessary for a domain expert to verify both the input and output of each of the phases of this algorithm. Once the classes have been identified and validated, the process of completing the GMDB is as in case 1.

4.2.3 Example of transformation algorithm

To illustrate the process translating from a relational database to a GMDB, let us consider an example of a simple domain. The relational database schema, with primary keys underlined, is self-explanatory.

EMP	=	[E#, Name, Salary, Deptno]
DEPT	=	[D#, Name]
PROJ	=	[Pname, Budget]
WORKS-ON	=	[E#, Pname]
DEPT-LOCS	=	[D#, Location]
MGR	=	[E#, Office]

We have the following constraints:
$EMP.DEPTNO$ is a foreign key of relation $DEPT$.
$WORKS-ON.E\#$ is a foreign key of relation EMP.
$WORKS{-}ON.Pname$ is a foreign key of relation $PROJ$.
There is an equality constraint between $DEPT.D\#$ and $DEPT-LOCS.D\#$.
$MGR.E\#$ is inclusion dependent on $EMP.E\#$.

The algorithm proceeds as follows:

Step 1: Three entity relations, EMP, $DEPT$, and $PROJ$ are identified.

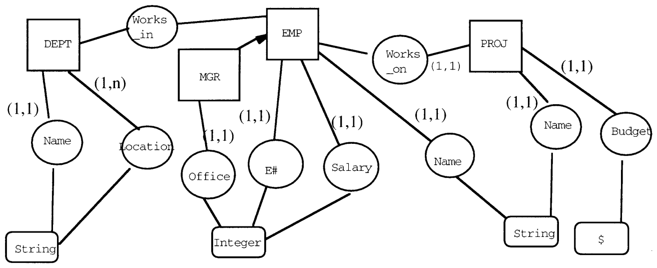


Fig. 7. Typed Graph of the resultant GMDB

Step 2: Two relationships are defined: case 1 identifies a relationship between EMP and $DEPT$, while case 2 identifies a many-to-many relationship between EMP and $PROJ$.

Step 3: This identifies $DEPT{-}LOCS.Location$ as a multivalued attribute of the entity relation $DEPT$.

Step 4: Case 1 identifies MGR as a subclass of EMP.

The Typed Graph of the resultant GMDB is shown in Fig. 7.

4.3 Object-oriented data models

Object-oriented data models are equipped with both powerful structural abstractions (e.g., generalization, classification, aggregation) that satisfy the representation requirements of the new kinds of applications (CAD, CAM, AI) and encapsulated procedures taking into account behavioral aspects. 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. No general agreement exists on the definition of the object-oriented data models. Anyhow we will concentrate on a number of concepts which 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]). 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.

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 identifier, 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 no limitation exists on the domain of attributes, i.e., an attribute domain can range on another class, the definition of a class may result 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 subclasses and superclasses. 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.

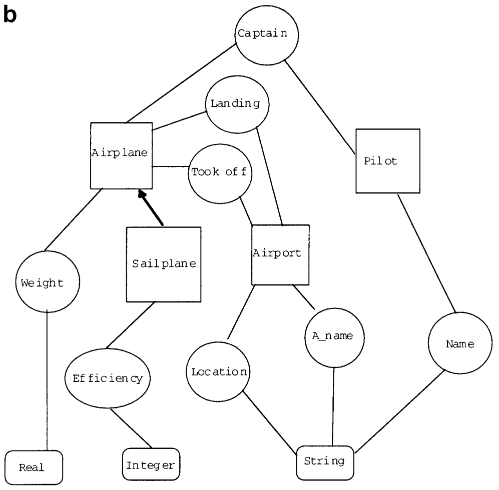
Summarizing, let $CN = \{cn_1, \ldots, cn_n\}$ be the set of class names, $DN = \{dn_1, \ldots, dn_n\}$ be the set of domain names, and let $D = \{D_1, \ldots, D_n\}$ be the family of *domains*, each composed by a set of *values* (we suppose to have finite domains); let $values$ be the function, $values : DN \to D$, associating a name to each domain; $OID = \{oid_1, \ldots, oid_k\}$ be the set of object identifiers, and $AN = \{an_1, \ldots, an_p\}$ be the set of attribute names. A class C is a 4-tuple $\langle cn, superclasses_{cn}, structure_{cn}, instances_{cn} \rangle$, where $cn \in CN$ is the class name, $superclasses_{cn}$ is the set of the names of the superclasses of C, $structure_{cn}$ is a set of pairs $\{\langle an_1, x_1 \rangle, \ldots, \langle an_{kcn}, x_{kcn} \rangle\}$ where for each pair $\langle an_i, x_i \rangle$ $an_i \in AN$ and either $x_i \in CN$ or $x_i \in DN$. $Instances_{cn}$ is a set made of kcn sets of tuples $\{inst_1, \ldots, inst_{kcn}\}$, one for each pair $\langle an_i, x_i \rangle$ belonging to $structure_{cn}$. Each tuple belonging to $inst_j$ has the following structure: $\langle cn : oid, an_j : y \rangle$, where $oid \in OID$ and y belongs either to OID or to D_j depending on the associate 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*).

Since the GM is object-based itself, the mapping $\mathcal{H}$ between object-oriented databases and GMDBs is almost straightforward. Basically, the family of domains D corresponds to the set of interpretations of the printable nodes of g (i.e., N_{C_p}), and the names of such domains are given by the labels of the elements of N_{C_p}. The set AN of attribute names is equal to the set of labels of the role-nodes (i.e., N_R) of g, and the meaning of such nodes corresponds to the associated set of tuples in the $instances$ of the classes the attributes belong to. The set CN of class names coincides with the set of labels of unprintable nodes in g, while the set of adjacent role-nodes is derived by the $structure$ of the class. The meaning of an unprintable node coincides with the set of the associated OIDs of the corresponding class. The inverse mapping can be easily defined as well. Printable and unprintable class-nodes correspond to domains and classes,

a

```
CLASS Airplane    [
   Weight:Real(1:1);{<Airplane:OI1,Real:200>,<Airplane:OI2,Real:4000>};
   Took_off:Airport;{<Airplane:OI1,Airport:OI4>,<Airplane:OI2,Airport:OI4>};
   Landing:Airport;{<Airplane:OI1,Airport:OI4>,<Airplane:OI2,Airport:OI5>};
   ]
CLASS Sailplane   [
   Superclasses:Airplane;
   Efficiency:Integer(1:1);{<Sailplane:OI1,Integer:38>};
   ]
CLASS Airport     [
   A_name:String(1:1);{<Airport:OI4,String:Kennedy>,<Airport:OI5,String:Fiumicino>};
   Location:String(1:1);{<Airport:OI4,String:NY>,<Airport:OI5,String:Rome>};
   ]
CLASS Pilot       [
   Name:String(1:1);{<Person:OI6,String:Steve>,<Person:OI7,String:Iacocca>,
                     <Person:OI8,String:Mark>,<Person:OI9,String:Kim>};
   Captain:Airplane;{<Person:OI6,Airplane:OI1>,<Person:OI7,Airplane:OI2>};
   ]
```

b



c

```
m (Pilot) = {OI6, OI7, OI8, OI9};
m (Airplane) = {OI1,OI2};
m (Sailplane) = {OI1};
m (Airport) = {OI4, OI5};
m (Integer) = Z;
m (Real) = R;
m (String) = {a...z}*;
m(Name)={<Person:OI6,String:Steve>,<Person:OI7,String:Iacocca>,
            <Person:OI8,String:Mark>,<Person:OI9,String:Kim>};
m(A_name)={<Airport:OI4,String:Kennedy>,<Airport:OI5,String:Fiumicino>};
m (Weight) = {<Airplane:OI1,Real:200>,<Airplane:OI2,Real:4000>};
m(Took_off)={<Airplane:OI1,Airport:OI4>,<Airplane:OI2,Airport:OI4>};
m(Landing)={<Airplane:OI1,Airport:OI4>,<Airplane:OI2,Airport:OI5>};
m(Captain)={<Person:OI6,Airplane:OI1>,<Person:OI7,Airplane:OI2>};
```

Fig. 8. An object-oriented database and the corresponding GMDB

while the set of role-nodes which are adjacent to a given unprintable class-nodes can be translated in terms of the class $structure$.

The example of mapping between an object-oriented database and a GMDB in Fig. 8 further illustrates the similarities between the two models. In Fig. 8a the object-oriented schema classes are described using a very intuitive syntax, while in Fig. 8b,c the corresponding Typed Graph and Interpretation are shown.

The object-oriented database contains pieces of information about flights and airplanes, including the source and destination airports of the flight and the name of the captain. In building the corresponding GMDB, the domains *Integer*, *Real*, and *String* have been translated into correspondent printable nodes; the class attributes *Weight*, *Efficiency*, *Location*, *A_name*, *Took_off*, *Landing*, *Captain*, and *Name* have been translated in terms of suitable role nodes; the classes *Airplane*, *Sailplane*, *Airport*, and *Pilot* have been translated into corresponding unprintable nodes. It is worth noting that the links among classes in the GMDB are not oriented as they were in the original model: class attributes ranging over other classes (e.g., the attribute *Took_off* of the class *Airplane*, ranging over the class *Airport*) are translated in terms of role-nodes linking both classes.

4.4 Semantic models

In this section we concentrate on semantic data models, which share with object-oriented models the availability of powerful structural abstractions. As two representative examples of this class we take 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, and the EER model [Chen 1976; Ullman 1987].

4.4.1 The IFO model

A IFO database ($IFODB$) 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.); (2) *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 model are 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 $dom(R) = \{\{O_1, \ldots, O_m\} | m \geq 0,$ and for each i, O_i is in $dom(R1)\}$.
3. If the root r of R is an aggregation vertex, and $R_1, \ldots, R_n$ are the ordered child subtrees of r, then $dom(R) = \{\langle O_1, \ldots, O_n\rangle |$ for each $i,\ O_i$ is in $dom(Ri)\}$.

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: $\langle r, a_1, r_1, a_2, r_2, \ldots, a_{n-1}, r_{n-1}, r_n\rangle$, 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 $f_1 = \langle r, p1\rangle, \ldots, f_n = \langle r, p_n\rangle$, be the functional edges of R with tail r; let R_O be the type with root r; and for each k, $k = 1 \ldots n$, 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 $\{f_1, \ldots, fn\}$ 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 $\mathscr{H}$ between the IFO databases and the GMDBs: $\mathscr{H}(IFODB) = D$, where $D = \langle g, m, c\rangle$ is a database expressed in terms of a Typed Graph g, an Interpretation m, and a set of constraints c. Since the basic idea of the IFO model is in building complex objects by recursively applying different type constructors, while the GM type set is limited to class-nodes and role-nodes, the mapping is quite complex and requires the generation of new unprintable class-nodes and role-nodes. The nesting of the IFO structures is expressed assigning an appropriate interpretation to the new role-nodes.

In order to specify the mapping $\mathscr{H}$, we make use of a function $\mathscr{T}$, defined on the instances of an IFO database. Let $x \in dom(R)$, if R is an atomic type, then $\mathscr{T}(x) = x$; else $\mathscr{T}(x) = new(x)$, where new is a function generating a new unprintable value when applied to a generic instance. The mapping $\mathscr{H}$ is specified as follows:

1. For each atomic type $v_i \in V_P$, there is a corresponding node $n_i \in N_{C_p}$ such that $f_1(n_i)$ is the name associated with v_i and $m(ni) = instances(v_i)$.

2. For each atomic type $v_i \in V_A \cup V_F$, there is a corresponding node $n_i \in N_{C_u}$ 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 $n \in N_{C_u}$, $n_j \in N_R$ and an edge $\langle n, n_j \rangle$, 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: $\langle f_1(n) : new(y), f_1(n') : \mathscr{T}(x) \rangle$, 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 $R_1, \ldots, R_l$, there are a node $n \in N_{C_u}$, such that $f_1(n)$ is the name associated with v, a node $n_j \in N_R$ and the edge $\langle n, n_j \rangle$ corresponding to the root v. Moreover, there are l recursively defined trees corresponding to the R_is whose roots n_i 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: $\langle f_1(n) : new(y), f_1(n_1) : \mathscr{T}(R_1), \ldots, f_1(n_l) : \mathscr{T}(R_l) \rangle$, where $y \in instances(v)$ and $x_i \in instances(R_i)$, $i = 1 \ldots l$.
5. For each functional edge $e = \langle r_0, x \rangle$ 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 $\langle n_1, n \rangle$, where n_1 is the root of the tree corresponding to the type with root r_0, and one edge $\langle n_2, n \rangle$, 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: $\langle f_1(n_1) : \mathscr{T}(x_0), f_1(n_2) : \mathscr{T}(x_1) \rangle$, where $\langle x_0, x_1 \rangle \in F(l)$.
6. For each ISA edge $i = \langle a, b \rangle$ 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 (or rule 2) says that IFO printable (or abstract and free) types are translated in terms of printable (or unprintable) class-nodes. Rule 3 (or rule 4) says that each association (or 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 (or 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.

The mapping $\mathscr{H}'$ from any GMDB to the corresponding IFO database (not detailed in the following), is easier than its inverse which is described above, since the GM components can be almost directly mapped into IFO constructs.

In Fig. 9 an example of mapping between an IFO database and the corresponding GMDB is drawn. Figure 9a shows the IFO schema of a database containing information about trips and foreign languages which will be spoken in each trip, guides and foreign languages they speak; Figure 9b contains the most relevant part of the corresponding extension. In Fig. 9c,d the corresponding Typed Graph and Interpretation are shown. Note that the association object t_langs, representing set of languages, has been translated in terms of the unprintable class-node t_langs, containing the identifiers of the sets of languages, and the role-node $R4$, associating each set with the languages it contains. An analogous translation has been applied to g_langs.

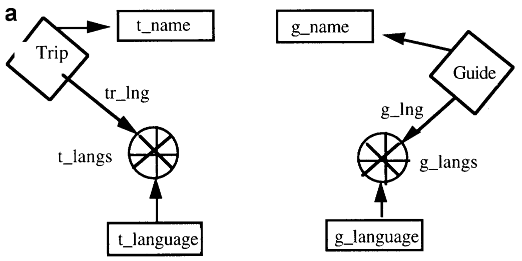
4.4.2 The entity-relationship model

An *Entity-Relationship database* ($ERDB$) is a collection of entity types, relationship types, and domains. Each *entity* is a "thing" in the real world, with an independent existence. Each entity has particular properties called *attributes*, that describe it. Each attribute is associated with a *domain*, which specifies the set of values that may be assigned to that attribute for each individual entity. Such entities define an *entity type*. Each entity type is characterized by a name and a list of attributes, forming the *entity type schema*, which is shared by the individual entities of that type.

The set of individual entity instances at a particular moment in time is called an *extension* of the entity type. In many cases an entity type will have numerous additional subgroupings of its entities, which are meaningful and need to be represented explicitly. We call each of these subgroupings a *subclass* of the entity type, and the entity type itself is called the *superclass* for each of these subclasses. We say that an entity that is a member of a subclass *inherits* all the properties of the entity as member of the superclass. A *relationship type* among n entity types is a set of associations among entities from these types. Each relationship instance is an association of entities, where the association includes exactly one entity from each participating entity type.

Summarizing, let $EN = \{en_1, \ldots, en_n\}$ be the set of *entity type names*, $AN = \{an_1, \ldots, an_p\}$ be the set of attribute names, $DN = \{dn_1, \ldots, dn_m\}$ be the set of *domain names*, $RN = \{rn_1, \ldots, rn_n\}$ be the set of *relationship type names*, $EID = \{eid_1, \ldots, eid_k\}$ be the set of entity identifiers, and $D = \{D_1, \ldots, D_m\}$ be the family of *domains*, each composed by a set of *values* (we suppose to have finite domains); let $values$ be the function, $values : DN \rightarrow D$, associating a name to each domain; let $assign$ be the function, $assign : EN \rightarrow$ Powerset(EID), associating a subset of the set of entity identifiers to each entity type name, and let $dom : AN \rightarrow D$ be a function associating a domain to each $an \in AN$. An entity type E is a 4-ple $\langle en, superclasses_{en}, structure_{en}, instances_{en} \rangle$, where $en \in EN$ is the entity type name, $superclasses_{en}$ is the set of the names of the superclasses of E, $structure_{en}$ is a set of attribute names $\{\langle an_1, \ldots, an_{ken} \rangle\}$ where each $an_i \in AN$; $instances_{en}$ is a set of sets of tuples, one for each element $e \in assign(en)$. The set of tuples asso-

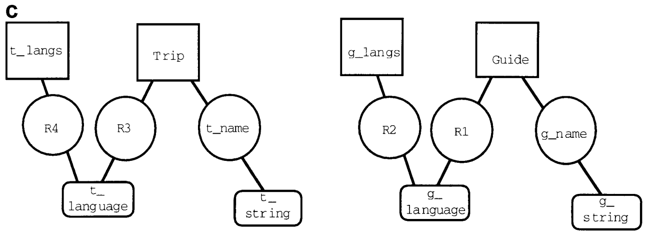
a



b

```
Instances (Trip) = {a, b, c}
Instances (t_name) = {Greece, China, France}
Instances (t_language) = {english, french, chinese, greek}
Instances (t_langs) = {{english, chinese}, {english, greek}, {french}}
tr_lng(Greece)/t_language = {english, greek}
tr_lng(China)/t_language = {english, chinese}
tr_lng(France)/t_language = {french}
Instances (Guide) = {x, y, z}
Instances (g_name) = {John, Mary, Leslie}
Instances (g_language) = {english, dutch, french, greek}
Instances (g_langs) = {{english, dutch}, {english, greek, french}, {english, french}}
tr_lng(John)/t_language = {english, greek, french}
tr_lng(China)/t_language = {english, dutch}
tr_lng(France)/t_language = {english, french}
```

c



d

```
m (Trip) = {a, b, c};
m (Guide) = {x, y, z};
m (t_langs) = {OI1, OI2, OI3};
m (g_langs) = {OI4, OI5, OI6};
m (t_language) = {english, french, chinese, greek};
m (t_string) = {Greece, China, France};
m (g_language) = {english, dutch, french, greek};
m (g_string) = {John, Mary, Leslie};
m(R4)={<t_langs:OI1,t_language: english>,<t_langs:OI1,t_language: chinese>,
            <t_langs:OI2,t_language: english}>, <t_langs:OI2,t_language: greek}>,
    <t_langs:OI3,t_language: french}>};
m (R3) = {<Trip:a, t_language: english>, <Trip:a, t_language: chinese>, <Trip:b, t_language:
            english>, <Trip:b, t_language: greek>, <Trip:c, t_language: french>};
m(t_name)={<Trip:a,t_string:China>,<Trip:b,t_string:Greece>, <Trip:c,t_string:France>};
m(R2) = {<g_langs:OI4,g_language: english>,<g_langs:OI4,g_language: dutch>,
            <g_langs:OI5,g_language: english}>, <g_langs:OI5,g_language: greek}>,
            <g_langs:OI5,g_language: french}>, <g_langs:OI6,g_language: french}>,
            <g_langs:OI6,g_language: english}>};
m(R1) = {<Guide:x, g_language: english>, <Guide:x, g_language: dutch>, <Guide:y,
            g_language: english>, <Guide:y, g_language: greek>, <Guide:y, g_language:
            french>, <Guide:z, g_language: english>, <Guide:z, g_language: french>};
            m(g_name)={<Guide:x,g_string:John>,<Guide:y,g_string:Mary>,
            <Guide:z,g_string:Leslie>};
```

Fig. 9a–d. The IFO database and the corresponding GMDB

ciated with e is $\{inst_1, \ldots, inst_{ken}\}$, with one $inst_i$ for each pair an_i belonging to $structure_{en}$. Each tuple belonging to $inst_j$ has the following structure: $\langle en : e, an_j : y\rangle$, where $y \in dom(an_j)$. A relationship type R is a 4-ple $\langle r_n, structure1_{rn}, structure2_{rn}, instances_{rn}\rangle$, where $rn \in RN$ is the relationship type name, $structure1_{rn}$ is a set of entity type names $\{en_1, \ldots, en_{krn}\}$, with each $en_i \in EN$, $structure2_{rn}$ is a set of domain names $\{dn_1, \ldots, dn_{hrn}\}$, with each $dn_i \in DN$. $instances_{rn}$ is a set of tuples of the form: $\langle en_1 : eid_1, en_{krn} : eid_{krn}, dn_1 : d_1, \ldots, dn_{hrn} : d_{hrn}\rangle$, where $en_1, \ldots, en_{krn} \in EN$, $eid_1, \ldots, eid_{krn} \subseteq assign(en_1) \times assign(en_2) \times \ldots \times assign(en_{krn})$ $dn_1, \ldots, dn_{hrn} \in DN$, $d1, \ldots, d_{hrn} \subseteq values(dn_1) \times values(d_2) \times \ldots \times values(dn_{hrn})$; note that $structure2_{rn}$ may be empty.

An *Entity-Relationship GM* database is a GMDB that satisfies the following constraint:
for each node n_i in N_R such that $|N_{C_u} \cap AD(n_i)| = 1$ it holds that $|N_{C_p} \cap AD(n_i)| = 1$ and $|AD(n_i)| = 2$. In other words, each role-node linked to only one unprintable class-node has to be involved in exactly another link, in particular with a printable class-node.

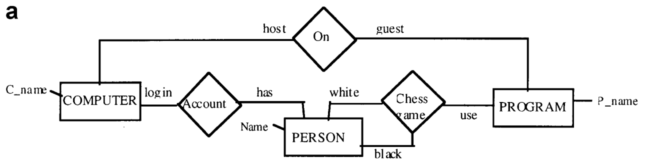
Let us define the mapping $\mathscr{H}$ between the ER model and the GM: if $ERDB$ is an ER database, then $\mathscr{H}(ERDB) = D$, where $D = \langle g, m, c\rangle$ is a database expressed in terms of a Typed Graph g, an Interpretation m, and a set of constraints c. The mapping $\mathscr{H}$ is specified as follows:

1. For each $dn_i \in DN$, there is a corresponding node $n_i \in N_{C_p}$ such that $f_1(n_i) = dn_i$ and $m(n_i) = values(dn_i)$.
2. For each $an_i \in AN$ there is a corresponding node $n_i \in N_R$ such that $f_1(n_i) = an_i$ and $m(n_i) = inst_i$, where $inst_i$ is the set of tuples associated with an_i in the structure of a class cn. Moreover, there is an edge $\langle n_i, n_j\rangle$ such that $n_j \in N_{C_p}$ is as defined in 1) and $m(n_j) = dom(an_j)$.
3. For each $en_i \in EN$ there is a corresponding node $n_i \in N_{C_u}$ such that $f_1(n_i) = en_i$ and its interpretation coincides with the set of entity identifiers of the corresponding entity type: $m(n_i) = assign(en_i)$; for each $s \in structure_{en_i}$ there is the edge $\langle n_i, f_1(s)\rangle$; moreover, for each $x \in superclasses_{en_i}$ there is the constraint $n_i \mathrm{ISA} x$.
4. For each $rn_i \in RN$ there is a corresponding node $n_i \in N_R$ such that $f_1(n_i) = rn_i$; there are k edges $\langle n_i, n_j\rangle$, $n_j \in N_{C_u}$, such that $\{f_1(n_1) \ldots f_1(n_k)\} = structure1_{rn_i}$ and h edges $\langle n_i, n_j\rangle$, $n_j \in N_{C_p}$, such that $\{f_1(n_1) \ldots f_1(n_h\} = structure2_{rn_i}$. Moreover, $m(n_i) = instances(rn_i)$.

Rule 1 states that the family of domains D corresponds to the set of interpretations of the printable nodes of g (i.e., N_{C_p}), and the names of such domains are given by the labels of the elements of N_{C_p}. Rule 2 states that the union of the sets AN of attribute names and RN of relationship type names is equal to the set of labels of the role-nodes (i.e., N_R) of g, and that in the first case the associated domains correspond to the interpretation of the adjacent printable nodes (note that each role-node corresponding to an attribute has a single adjacent printable node). In the second case the $structure1_{rn}$ (or $structure2_{rn}$) of the relationship type corresponds to the set of labels of the adjacent unprintable (or printable) class-nodes. Rule 3 states that the set EN of entity type names coincides with the set of labels of unprintable nodes in g, while the set of adjacent role-nodes is derived by the $structure$ of the class. The meaning of an unprintable node coincides with the set of EIDs of the corresponding class.

In order to better clarify the above mapping we furnish an example of translation for the ER schema shown in Fig. 10a,b, representing information about network chess games in progress where we use a very intuitive syntax for characterizing the schema extension. The resulting GMDB is shown in Fig. 10c. Note that, since the GM does not support reflexive role nodes, a new unprintable class node has been introduced (*Chess game*).

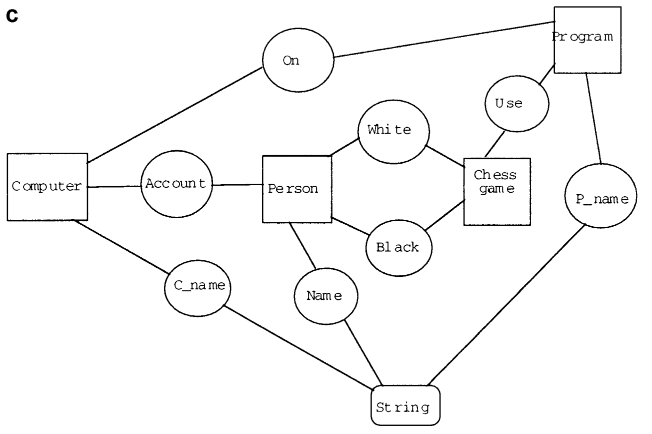
a



b

```
M(COMPUTER)={<Infokit>, <Assi>, <Athena>}
M(PERSON)={<John>, <Mary>, <Peter>,<Carl>, <Cecily>, <Carol>}
M(PROGRAM)={<InterChess>, <ChessG>, <CsG>}
M(ACCOUNT)={<login:Infokit,has:John>,<login:Assi,PE,has:Mary>,<login:Assi,has:Peter>,
            <login:Assi,,has:Carl>,<login:Assi,has:Cecily>}
M(ON)={<host:Infokit,guest:InterChess>, <host:Infokit,guest:ChessG>, <Computer:OAssi,Program:OCG>}
M(CHESSGAME)={<white:John,black:Peter,use:InterChess>, <white:Mary,black:Carl,use:ChessG>,
              <white:Carl,black:Cecily,use:InterChess>}
```

c



```
m(COMPUTER)={<O1>, <O2>, <O3>}
m(PERSON)={<O4>, <O5>, <O6>,<O7>, <O8>, <O9>}
m(PROGRAM)={<O10>, <O11>, <O12>}
m(CHESS GAME)={<O13>, <O14>, <O15>}

m(ACCOUNT)={<Computer:O1,Person:O4>, <Computer:O2,Person:O5>, <Computer:O2,Person:O6>,
            <Computer:O2,Person:O7>,<Computer:O2,Person:O8>}
m(ON)={<Computer:O1,Program:O10>, <Computer:O1,Program:O11>, <Computer:O2,Program:O12>}
m(WHITE)={<Person:O4,Chess game:O13>, <Person:O5, Chess game:O14>,<Person:O7,Chess game:O15>}
m(BLACK)={<Person:O6, Chess game:O13>,<Person:O7,Chess game:O14>, <Person:O8, Chess game:O15>}
m(USE)={<Program:O10,Chess game:O13>,<Program:O11,Chess game:O13>,
            <Program:O10,Chess game:O13>}
m(STRING)={a..z}*;
```

Fig. 10a–c. The chess game schema and the corresponding GMDB

5 Query management

The aim of this section is twofold. We first describe how queries expressed in relational, object-oriented, and semantic languages can be defined in terms of GPs; then, we show how it is possible to translate a GP query in terms of a relational language, in order to be processed by the actual relational DBMSs. Here we concentrate on relational DBMSs, since they are widely diffused in real applications. However, the definition of the mappings with object-oriented and semantic data models, as well as the inner structure of the GM and the GPs, will permit easy extension of our approach to the effective interfacing of object-oriented and semantic DBMSs.

5.1 Expressing relational queries

In [Catarci Santucci Angelaccio 1993] we dealt with the expressive power of the GPs, proving that the class of queries computed by the GPs contains the class of queries computable by the Relational Algebra. In this section we give an example of translating a relational query in terms of GPs.

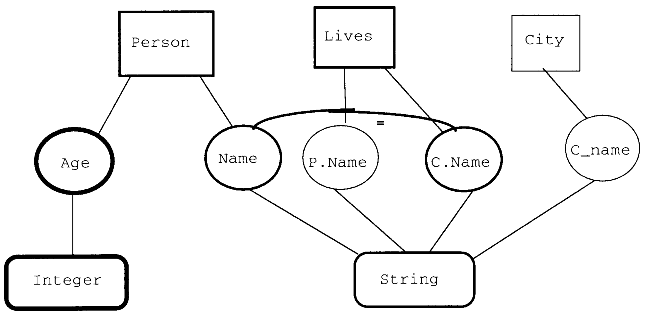


Fig. 11. Expressing a relational query through the GPs

Referring to the example of Sect. 4.2.3, assume the relational expression is as follows:

$$\Pi_{\text{Age}}(\sigma_{\text{Name=C.name}}(PERSON \times LIVES))$$

It corresponds to the query: find out the age of the people whose name is equal to the name of a city in which some other person lives.

As we have shown in [Catarci Santucci Angelaccio 1993], the projection on some attributes corresponds to set to `displayed` the nodes corresponding to the attributes; the join (cartesian product plus selection) is rendered through the drawing a new labeled edge. The GPs corresponding to the above relational expression are shown in Fig. 11.

5.2 Expressing object-oriented queries

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 equipped with formally defined query languages (relational algebra and calculus) that constitute a well-known expressive power yardstick, object-oriented proposals lack a widely accepted 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 different choices available for determining the structure of the result of a query on an object-oriented database. Indeed, the presence of class hierarchies leads to two possible interpretations for the access scope of a query, in particular, when the target class c of the query is a superclass of other classes in the schema:

– The scope is the set of objects that are instances of c, either directly or indirectly through ISA relations.
– The scope is the set of objects that are instances of c but are not instances of any other subclass of c.

In order for the result to be constituted by a set of objects that are all characterized by a fixed number of attributes, we have to either choose the second approach, or to follow the first one, with the proviso that attributes of the subclasses of c are not included in the result. In our work, we indeed adopt the first approach with the above mentioned proviso.

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], which we now briefly recall. Object-oriented queries are classified into 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 conditions on its attributes. The domain of the involved attributes determines two possible kinds of conditions: 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 $\langle attribute - name, operator, value\rangle$, 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. Note that a single-operand query is more powerful than a single-relation query, because it involves joins of several classes when we make use of complex predicate. More precisely, an implicit join is applied when we retrieve objects from the target class by referring to 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.

We note that single-operand queries are not sufficient for reaching the expressive power of relational algebra; indeed, at least explicit joins of classes on user-specified attributes, as well as set-operations between classes, must be added to this purpose. 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 they 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 always 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 descendants of the same parent, thus having compatible types. 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 make use of examples in order to show how the GPs allows us to express object-oriented queries. Note that we refer to the object-oriented database in Fig. 8a and the corresponding GMDB in Fig. 8b,c. Let us consider the query "Find the weight of all the sailplanes whose efficiency is greater than 30 and whose landing airport is NY." This query falls in the class of single operand queries, as defined above, and involves both a simple (i.e., $Sailplane.Efficiency > 30$) and a complex (i.e., $Sailplane.Landing.A\_name$ = "NY") predicates. The corresponding query in the language shown in [Kim 1989] is: select $Sailplane(Efficiency > 30$ and $Landing.A\_name$ ="NY"). In our approach such a query is expressed by first selecting the nodes $Sailplane$, $Efficiency$, $Integer$, $Landing$, $Airport$, A_name, $String$, and setting to displayed the nodes $Weight$ and $Real$; then, changing the labels of the edges $\langle Efficiency, Integer\rangle$ and $\langle A\_name, String\rangle$ into $Efficiency.Integer > 30$ and $A\_name.String$ ="NY", 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 all the airports on which an airplane is landing whose captain's name is included in the airport name". The corresponding query, if reversible pointers are available, can be expressed in the language shown in [Kim 1989] as: select $Airport(Landing.Captain.\ Name$ IN $A\_name)$; if the model does not include such kind of pointers the query must be expressed as follows: select $Pilot(Name$ IN $Airplane.\ Landing.A\_name)$ and we can access the desired result (i.e., airport names) as an attribute of $Pilot$. 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 optimisation (see, e.g., [Ullman 1987]). The above query is expressed by first selecting the nodes $Airport$, $Pilot$, $Airplane$, $Captain$, $Landing$, $Name$ and setting to displayed the nodes A_name and $String$; then drawing an edge between nodes $Name$ and A_name labelled with $Name.String$ IN $A\_name.String$. Note that the drawing of an edge corresponds to an explicit join, as mentioned above.

5.3 Expressing semantic queries

In Sect. 4.4 we translated IFO and ER databases in terms of GMDBs. An important feature of both the IFO and the ER models is in having associated graphical query languages, namely SNAP [Bryce Hull 1986], and QBD [Angelaccio Catarci Santucci 1990]. We can exploit the existence of such languages for showing not only how to express the queries of a semantic query language in terms of our GPs, but also for demonstrating that the GPs can be easily used to precisely characterize the semantics of other grap hical languages.

5.3.1 Expressing SNAP queries

Since the syntax used in SNAP for representing IFO data schemata is defined in terms of different graphical elements (circles, diamonds, labels, edges, and connection rules), we need to provide a mapping between the syntax of SNAP and the syntax of the GM. Such a mapping, called s, between SNAP diagrams (SNAPD) and Typed Graphs is defined as follows:

1. For each rectangle $r\langle\in\rangle SNAPD$ with label lr, there is a node $n_i\langle\in\rangle N_{C_p}$ such that $f_1(n_i) = lr$.
2. For each diamond $rh\langle\in\rangle SNAPD$ with label lh, there is a $node$ $n_i\langle\in\rangle N_{C_u}$ such that $f_1(n_i) = lh$.

3. For each circle $c\langle\in\rangle SNAPD$ with label lc, there is a node $n_i\langle\in\rangle N_{C_u}$ such that $f_1(n_i) = lc$.
4. For each star circle[2] $sc \in SNAPD$ with label lsc, there is a pair of linked nodes, $n_i \in N_{C_u}$ and $n_j \in N_R$, such that $f_1(n_i) = lsc$.
5. For each cross circle[3] $pc \in SNAPD$ with label lpc, there is a pair of linked nodes, $n_i \in N_{C_u}$ and $n_j \in N_R$, such that $f_1(n_i) = lpc$.
6. For each unarrowed edge $\langle x, y\rangle \in SNAPD$ (x is parent of y in a tree), there is an edge $\langle x', y'\rangle \in E$ such that $x' \in s(x)$[4] and $y' \in s(y)$[5].
7. The set of arrowed edges belonging to a functional path p of length n, is translated into a role-node z and $n+1$ edges of the form $\langle z, \mathscr{S}(r_i)\rangle$, $i = 0 \ldots n$, with r_i defined as above.

By using the mappings $\mathscr{H}$, defined in Sect. 4.4.1, and 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, and *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.

Such operations, applied to an IFO database $IFODB$, whose visual representation is a SNAP diagram $SNAPD$, correspond to several applications of GPs on the database $D = \langle g, c, m\rangle = \mathscr{H}(IFODB)$ such that $g = s(SNAPD)$. 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 $n(s(c_1)) \ldots n(s(c_k))$) that are generated translating the tree components $c_1 \ldots c_k$, which are nodes, edges, functional paths:

$$s(\mathscr{H}(IFODB), n(s(c_1))) \circ \ldots s(\mathscr{H}(IFODB), n(s(c_k))).$$

2. The change of the label of a rectangle c with a new label $\mathscr{F}$ is expressed as the change of the label of the edge linking $s(c)$ to the (unique[6]) role-node q with a new label $\mathscr{F}'$, which corresponds to the GP syntax:

$$\mathscr{C}(\mathscr{H}(IFODB), \mathscr{F}', s(c), q).$$

3. The shading of a node p corresponds to setting to `displayed` the corresponding node in the GMDB:

$$s(\mathscr{H}(IFODB), s(p)).$$

4. The drawing of a labeled edge (with label $\mathscr{F}$) 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 $\mathscr{F}'$, which corresponds to drawing an edge between the two (unique) role-nodes n_1 and n_2, linked with the printable nodes $s(l_1)$ and $s(l_2)$: $\mathscr{E}(\mathscr{H}(IFODB), \mathscr{F}', n_1, n_2)$.
   - 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 there is a huge semantic distance between such operations and the GPs. For the sake of brevity, we do not give the detai this correspondence.

5.3.2 Expressing queries in QBD

QBD [Angelaccio Catarci Santucci 1990] is primarily a navigational language on Entity-Relationship (ER) diagrams representing conceptual schemata. The user first interacts with the conceptual schema to understand its information content, and extracts the subschema of interest containing the concepts involved in the query, then, during the "navigation" activity, he may express the query, defining all its procedural characteristics. Initially s/he selects a central concept, called *main concept*, that can be seen as the entry point of the query, then s/he can choose between two kinds of primitives for navigating in the schema. The first one allows the user to follow paths of concepts, the other one is used for comparing two concepts which are not directly connected to each other. Conditions on attributes are expressed by means of a window, where the list of the attributes is shown together with the elements involved in the comparison (i.e., constants, other attributes, etc.) and a set of icons suitable

2 Circles with an inscribed star are the representation for association nodes

3 Circles with an inscribed cross are the representation for aggregation nodes

4 $s(x)$ produces a set of nodes and x' is the role-node among them

5 If $s(y)$ produces more than one node, y' is the class-node

6 Note that the uniqueness of such a role-node is a consequence of the IFO definition of fragment, not a GM constraint

to formulate conditions on the attributes. Conditions are expressed selecting the attributes and the icon corresponding to the required operator.

Since the QBD syntax is defined in terms of various graphical elements (rectangles, circles, diamonds, labels, and edges), we also provide a mapping between the elements of QBD diagrams (QBDD) and the representation structures of Typed Graphs.

The mapping s is defined as follows:

1. For each rectangle $r \in QBDD$ with label lr, there is a node $n_i \in N_{C_u}$ such that $f_1(n_i) = lr$.
2. For each diamond $rh \in QBDD$ with label lh, there is a node $n_i \in N_R$ such that $f_1(n_i) = lh$.
3. For each circle $c \in QBDD$ with label lc, there is a node $n_i \in N_R$ such that $f_1(n_i) = lc$.
4. For each edge $\langle x, y \rangle \in QBDD$, there is an edge $\langle x', y' \rangle \in E$ such that $x' = s(x)$ and $y' = s(y)$.

By using the mappings $\mathscr{H}$ (see Sect. 4.4.2) and s we show that the query language QBD can be formalized in terms of GPs.

Let Q(QBD) be the set of queries expressible by using the graphical operations of QBD. Such graphical operations, introduced above, are:

1. Selection of the first rectangle (main entity)
2. Selection of any path diamond-rectangle
3. Selection of two disconnected rectangles
4. Opening of a window containing the labels of the circles, selection of some labels
5. Opening of a window containing the labels of the circles, drawing of an edge between some label, with a new label
6. Selecting two isolate rectangles and selecting of an icon representing a set operator

Such operations, applied to an ER database ERD, whose visual representation is a QBD diagram QBDD, correspond to several applications of GPs on a database $D = \langle g, c, m \rangle$ in the ER Graph Model, such that $D = \mathscr{H}(ERD)$ and $g = s(QBD)$. 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 the first rectangle r is expressed as: the selection of the node with the same label; the selection of all the nodes linked to it and corresponding to circles (denoted with $s(r_1) \ldots s(rk)$):

   $\mathscr{S}(\mathscr{H}(ERD), s(r)) \circ s(\mathscr{H}(ERD), s(r_1))$
   $\circ \ldots s(\mathscr{H}(ERD), (r_k))$.

2. The selection of any path rhomb-rectangle (denoted with $rh - r$) is the selection of the corresponding nodes:

   $\mathscr{S}(\mathscr{H}(ERD), s(r)) \circ s(\mathscr{H}(ERD), s(r_h))$.

3. The selection of two disconnected rectangles, say r_1, r_2, is the selection of the corresponding nodes:

   $\mathscr{S}(\mathscr{H}(ERD), s(r_1)) \circ s(\mathscr{H}(ERD), s(r_2))$.

4. The opening of a window containing labels of circles and the selection of some labels, say $lc_1 \ldots lc_h$, corresponds to the selection on the Typed Graph g of the role-nodes n_i such that

   $f_1(n_i) = lc_i : \mathscr{S}(\mathscr{H}(ERD), n_1) \circ s(\mathscr{H}(ERD), n_2)$
   $\circ \ldots s(\mathscr{H}(ERD), n_h)$.

5. The opening of a window containing labels of circles, say $lc_1 \ldots lc_h$, and the drawing of a labeled edge (with label $\mathscr{F}$) between two of them, lc_1, lc_2, correspond to the drawing of an edge, with the same label $\mathscr{F}$, between the two role-nodes n_1 and n_2 such that:

   $f_1(n_1) = lc_1$ and $f_1(n_2) = lc2 : \mathscr{E}(\mathscr{H}(ERD), \mathscr{F}, n_1, n_2)$.

6. The selection of two isolate rectangles and the selection of an icon representing a set operator, correspond to the drawing of an edge, with a label $\mathscr{F}$ equivalent to the icon, between the two nodes n_1 and n_2 corresponding to the rectangles:

   $\mathscr{E}(\mathscr{H}(ERD), \mathscr{F}, s(r_1), s(r_2))$.

As an example, let us consider the ER chess game schema of Fig. 10a and assume that we are interested in finding out the name of all the white players of a chess game together with the name of the computer on which they have an account. The corresponding ER query is expressed through the selection of the diagram elements constituting the path:

$\langle Computer, Account, Person, (white\ role), ChessGame \rangle$,

asking for the inclusion in the final result of the attributes $Name$ and C_name. Applying the above mapping rules, we obtain the GPs shown in Fig. 12.

5.4 Generating relational expressions

We showed in the previous section that the last phase of the query expression is devoted to the production of the query result expressed through a new GMDB, namely the result database D°. In this section we see how to directly build a relational algebra expression, representing the query result, which can be processed by the underlying relational DBMS. Moreover, we will prove that applying the inverse mapping rules to such a relational expression we obtain a GMDB belonging to the same equivalence class as D°.

In Sect. 4.2 we introduced suitable relational algebra expressions in order to give the role-node interpretation, i.e., for each $r \in N_R$ $RelExp(r)$ returns the relational algebra expression used in defining $m(r)$. In this subsection we show how to compose such relational algebra expressions in order to express the query result.

Note that we can associate the notion of $RelExp$ to the mapping from the relational model with no constraints to the GM (see Sect. 4.1). In this case, applying $RelExp$ to a role-node returns the label of the unprintable node adjacent to it, i.e., the associated relation name. Under this condition, the expression defined below computes a correct query.

Let D be an admissible GMDB[7] possibly containing unselected, selected and displayed nodes, together with edges with labels $\neq T$.

[7] Depending on the structure of the GMDB D, the result database may have an empty set of role-nodes. In this case we say that D *is not admissible*, otherwise we say that D *is admissible* (see [Catarci Santucci Angelaccio 1993], where the constraints a GMDB must satisfy in order to be admissible are specified)

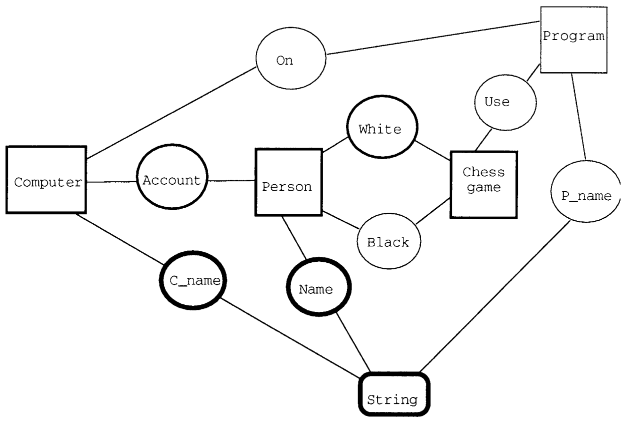


Fig. 12. Translating an ER query into GPs

Let $N_{rs}, N_{rd} \subseteq N_R$ be the set of selected and displayed role-nodes respectively. Let E_f be the set of edges whose label is $\neq T$. In order to compute the query result we compute the following relational algebra expression:

$$R = \Pi_P \left(\sigma_F \left(RelExp(n_1) \underset{f_{1,2}}{\bowtie} RelExp(n_2) \ldots \underset{f_{k-1,k}}{\bowtie} RelExp(n_K) \right) \right)$$

where $n_i \in N_R$, $f_3(n_i) \in \{\mathtt{selected}, \mathtt{displayed}\}$ for $i = 1 \ldots K, P, F$, and the equi-join conditions $f_{i,i+1}$ are defined as follows.

P is the set of the `displayed` role-nodes, $P = N_{rd}$;
F is the logical conjunction of the labels belonging to the edges in E_f;
$f_{i,i+1}$ is defined as follows:

1. If $RelExp(n_i)$ and $RelExp(n_{i+1})$ share the same primary key, i.e., $AD(n_i) \cap N_{C_u} = AD(n_{i+1}) \cap N_{C_u} = n$, then $f_{i,i+1}$ is equal to
   "$PK_{RelExp(n_i)} = PK_{RelExp(n_{i+1})}$".
2. If $RelExp(n_i)$ and $RelExp(n_{i+1})$ share the same foreign key, i.e., $AD(n_i) \cap AD(n_{i+1}) = n \in N_{C_u}$, then $f_{i,i+1}$ is equal to
   "$FK_{RelExp(n_i),RelExp(n)} = PK_{RelExp(n_{i+1}),RelExp(n)}$".
3. If there exists a foreign key of $RelExp(n_i)$ in $RelExp(n_{i+1})$, i.e., $AD(n_i) \cap AD(n_{i+1}) \cap N_{C_u} = n$ and $|AD(n_{i+1}) \cap N_{C_u}| > 1$, then $f_{i,i+1}$ is equal to
   "$FK_{RelExp(n_i),RelExp(n_{i+1})} = PK_{RelExp(n_{i+1})}$".
4. If none of the three above cases holds then $f_{i,i+1}$ is equal to $\emptyset$ and the operation performed is a cartesian product.

Theorem 1
Let D be an admissible GMDB obtained by applying a set of GPs to $D' = \mathscr{H}(DB)$ where DB is an enriched relational database. Let D° be the corresponding result database, and let R be the relation computed by the above relational expression associated with D. Let $D'' = \mathscr{H}(R)$ where R is the relational database composed by the single relation R. Then D'' is equivalent to D°, i.e., $D'' \in EQ(D^\circ)$.

Proof (sketch)
The proof proceeds by comparing the relational expression introduced in this section with the $eval$ operator introduced when defining the construction of D° (see Sect. 3.2.2). In fact, it is easy to show that $eval$ operator computes either a cartesian product or an equi-join on the unprintable values. The former corresponds to case 4 of the above relational expression computation, the latter to one of the first three cases, resulting in an equi-join on keys. The unprintable values of D° are new invented values and the corresponding ones in D'' are generated by the mapping $\mathscr{H}$. Therefore, the tuples in D'' can differ from the ones in D° only for the unprintable values and, consequently, D'' and D° are in the same equivalence class.

6 Summary and conclusions

This paper summarizes a consistent part of the work done during the past 5 years with the main purpose of designing a new VQS providing the user with a multiparadigmatic visual interface realizing an integrated access to heterogeneous databases. Such a VQS has two main components: (1) the user interface (described in [Catarci Chang Santucci 1994]), which is able to automatically adapt to different users by offering to them the most appropriate visual representation and interaction modality on the basis of a user model describing the user skill and needs; and (2) the interface to the different DBMSs, which allows for expressing the diverse databases in terms of a single model and eventually build an integrated schema. Both components rely on a formal model, the Graph Model (GM), having a graphical syntax and an object-based semantics, which plays the role of both unifying model for the heterogeneous components and intermediate visual representation used for formalizing the different final representations presented to the user. The GM has associated a query language composed by two Graphical Primitives (GPs): the selection of a node and the drawing of an edge. Again, the purpose of such primitives is twofold. They have the same expressive power of the query languages associated with the most common data models, and are a means for formalizing more complex visual interactions, as provided in the user interface.

The lower part of the system has been presented in this paper. In particular, we discussed how databases expressed in the most common data models can be translated in terms of GMDBs, possibly by exploiting reverse engineering techniques, and showed that the GPs are equivalent to well known query languages. Finally, we described how queries expressed by using the GPs can be translated in terms of relational expressions so to be processed by one type of actual DBMSs. We concentrated on relational DBMSs, since they are widely diffused in real applications. However, the definition of the mappings with object-oriented and semantic data models, as well as the inner structure of the GM and the GPs, permits easy extension of our approach to effectively interface object-oriented and semantic DBMSs.

As we said in the Introduction, our proposal differs from others available in the literature mainly in 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 papers present specific VQSs, which are intended to be directly utilized by the users. Our proposal presents a more general environment, where various equivalent user interfaces can be defined on the basis of the same formalisms, and different databases can be dealt with in a uniform way. Our proposal shares several similarities with Mark's approach both in the data model (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 are used instead of graphical operations. 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.

Summarizing, this research has provided a strong and formal basis for the development of an adaptive user interface to heterogeneous databases. The work continues. In particular, we are devising a more sophisticated schema integration technique. Starting from a set of GMDBs and a knowledge base containing intra- and inter-schema knowledge (expressed by using an extension of the Constraint Language) we will be able to automatize several phases of the schema integration. As for the multiparadigmatic interface, which has been implemented, we are presently testing it against real users.

Acknowledgements. The authors would like to thank the anonymous referees for their helpful comments. Moreover, Tiziana Catarci and Giuseppe Santucci are grateful to Maurizio Lenzerini for his valuable suggestions.

References

1. S. Abiteboul, R. Hull (1987) IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems* **12**: 525–565
2. J.R. Abrial (1974) Data Semantics. In: *Database Management*, North Holland, Amsterdam, pp. 1–59
3. M. Angelaccio, T. Catarci, G. Santucci (1990) QBD*: A Graphical Query Language with Recursion. *IEEE Transactions on Software Engineering* **16**: 1150–1163
4. F. Bancilhon (1988) Object-Oriented Database Systems. In: Proc. of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, Austin, Texas
5. F. Bancilhon, S. Cluet, C. Delobel (1990) A Query Language for the O2 object-oriented Database System. In: Proc. of the Second International Workshop on Database Programming Languages
6. J. Banerjee et al. (1988) OQL: An object-oriented Query Language. in Proc. of the 4th Intl. Conf. on Data Engineering, Los Angeles
7. C. Batini, M. Lenzerini (1984) A methodology for data schema integration in the Entity Relationship model. *IEEE Transactions on Software Engineering* **10**: 650–664
8. C. Batini, M. Lenzerini, S. Navathe (1986) A Comparative Analysis of Methodologies for database Schema Integration. *ACM Computing Surveys* **15**: 323–364
9. C. Batini, T. Catarci, M.F. Costabile, S. Levialdi (1991) Visual Query Systems. Technical Report N.04.91 of Dipartimento di Informatica e Sistemistica, Universita' di Roma "La Sapienza"
10. C. Beeri (1990) A Formal Approach to Object-Oriented Databases. *Data & Knowledge Engineering* **5**: 353–382
11. R.J. Brachman (1983) What IS-A is and isn't: An Analysis of Taxonomic Links in Semantic Networks. *IEEE Computer* **16**: 30–36,
12. R.J. Brachman, J.G. Schmolze (1985) An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science* **9**: 171–216
13. Y. Breitbart, H. Garcia-Molina, A. Silberschatz (1992) Overview of Multidatabase Transaction management. *VLDB Journal* **1**, 2
14. Briand et al. (1987) From Minimal Cover to E/R Diagram. Proc. of the 6th Entity-Relationship Conf., New York
15. D. Bryce, R. Hull (1986) SNAP: A Graphics-based Schema Manager. Proc. of the IEEE Conference on Data Engineering, Los Angeles, USA, pp. 151–164
16. M. Castellanos, F. Saltor (1991) Semantic Enrichment of Relational Schemas. Proc. of the IMS-91, Kyoto
17. T. Catarci, S.K. Chang, M.F. Costabile, S. Levialdi, G. Santucci (1993) A Multiparadigmatic Visual Environment for Adaptive Access to Databases. Proc. of the 1993 Conference on Human Factors in Computing Systems, INTERACT' 93 and CHI' 93, Amsterdam
18. T. Catarci, S.K. Chang and G. Santucci (1994) Query Representation and Management in a Multiparadigmatic Visual Query Environment - *Journal of Intelligent Information Systems*, Special Issue on "Advances in Visual Information Management Systems" **3**: 299–330
19. T. Catarci, G. Santucci, M. Angelaccio (1993) Fundamental Graphical Primitives for Visual Query Languages. *Information Systems* **18**: 75–98
20. S.K. Chang (1990) A Visual Language Compiler for Information Retrieval by Visual Reasoning. *IEEE Transactions on Software Engineering*, Special Section on Visual Programming, pp. 1136–1149
21. P.P. Chen (1976) The Entity-Relationship Model toward a Unified View of Data. *ACM Transactions on Data Base Systems* **1**, 1
22. E.F. Codd (1970) A Relational Model for Large Shared Data Banks. *Communications of the ACM* **13**, 6
23. E.F. Codd (1972) Relational completeness of database sub-languages. In: *Data Base Systems*, R. Rustin (ed.), Prentice Hall, Englewood Cliffs, pp. 65–98
24. M. Consens, A.O. Mendelzon (1990) Graphlog: A Visual Formalism for Real Life Recursion. Proc. of the ACM Symp. on Principles of Database Systems, pp. 404–416
25. I. Cruz (1990) Declarative Query Languages for Object-Oriented Databases. Technical Report CSRI-238, Computer Systems Research Institute, University of Toronto,
26. I.F. Cruz, A.O. Mendelzon, P.T. Wood (1988) G+: Recursive Queries Without Recursion. Proc. of the 2nd Int. Conf. on Expert Database Systems, pp. 355–368
27. C.J. Date (1987) *An Introduction to Database Systems*, Vol. I, Addison-Wesley Publishing Company
28. J. Davis, F. Arora (1987) Converting a Relational Database Model into an Entity Relationship Model. Proc. of the 6th Entity-Relationship Conf., New York
29. A. Elmagarmid, C. Pu (1990) Guest Editors Introduction to the Special Issue on Heterogeneous Databases. *ACM Computing Surveys* **22**, 3,
30. R. Elmasri, S.B. Navathe (1989) *Fundamentals of Database Systems*. Benjamin-Cummings Pub.
31. W. Gottard, P. Lockemann, A. Neufeld (1992) System-Guided View Integration for Object-Oriented Databases. *IEEE Transactions on Knowledge and Data Engineering* **4**: 1–22
32. M. Gyssens, J. Paredaens, D. Van Gucht (1990) A Graph-Oriented Object Model for Database End-User Interfaces. Proc. of the ACM SIGMOD Conference on the Management of Data, Atlantic City, USA, pp. 24–33
33. R. Hull, R. King (1987) Semantic Database Modeling: Survey, Applications and Research Issues. *ACM Computing Surveys* **19**: 201–260
34. B. Johanssen B. et al. (1989) Abstracting Relational and Hierarchical Data with a Semantic Data Model. In: Proc. of the 8th Entity-Relationship Conf., Toronto
35. K. Kalman (1989) Implementation and Critique of an Algorithm which Maps a Relational Database into Conceptual Schemas. In: Proc. of the 8th Entity-Relationship Conf., Toronto

36. W. Kim (1989) A Model of Queries for Object-Oriented Databases. Proc. of the 15th Intl. Conf. on Very Large Database, Amsterdam
37. W. Kim (1990) *Introduction to Object-Oriented Databases*. Computer Systems Series, The MIT Press, USA
38. K.C. Kim, W. Kim, A. Dale (1989) Cyclic Query Processing in Object Oriented Databases. In: Proc. of the 5th Intl. Conf. on Data Engineering, Los Angeles,
39. T. Landers, R. Rosenberg (1982) An overview of MULTIBASE. In: *Distributed Databases*, North Holland
40. L. Mark (1989) A Graphical Query Language for the Binary Relationship Model. *Information Systems* **14**: 231–246
41. U. Nanni (1988) A Graphic Interface for Relational Databases. Technical Report, Dipartmento di Informatica e Sistemistica, Universita' di Roma "La Sapienza", Roma, Italy
42. S.B. Navathe, S.G. Gadgil (1982) A Methodology for View Integration in Logical Database Design. Proc. of the 8th International Conference on Very Large Data Bases, Mexico City, pp. 142–164
43. S.B. Navathe, et al. (1987) A Federated Architecture for Homogeneous Information Systems. Workshop on Heterogeneous Databases
44. A. Sheth, J. Larson (1990) Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys* **22**, 3
45. D. Shipman (1981) The functional Data Model and the Data Language DAPLEX. *ACM Trans. on Database Systems* **6**, 1
46. B. Shneiderman (1983) Direct Manipulation: A Step beyond Programming Languages. *IEEE Computer* **16**: 57–69
47. G. Thomas, et al. (1990) Heterogeneous Distributed Database Systems for Production Use. *ACM Computing Surveys* **22**, 3
48. D.C. Tsichritzis, F. Lochovsky (1982) Data Models. Prentice Hall
49. J.D. Ullman (1987) *Principles of Database and Knowledge-Base Systems*, vol. I, Computer Science Press, Rockville, MD
50. Yan, L-L. (1992) Deriving an Object Oriented Schema from a Relational Schema. IFIP-DS5 Conference, Australia
51. M.M. Zloof (1977) Query-by-Example: A Database Language. *IBM Syst. Journal* **16**: 324–343