QBD*: A Fully Visual System for E-R Oriented Databases

M. Angelaccio
Dipartimento di Ingegneria Elettronica
Università degli Studi di Roma II "Tor Vergata"
Via O. Raimondo 38- 00173 Roma, Italy

T. Catarci G. Santucci
Dipartimento di Informatica e Sistemistica
Università degli Studi di Roma "La Sapienza"
Via Buonarroti 12 - 00185 Roma, Italy

ABSTRACT
The need of a friendly man-machine interaction is becoming crucial for a large variety of applications, in particular, those requiring frequent extraction of information from the database. Experience suggests that traditional query languages are not friendly enough for the casual user: she/he is requested to formulate queries in a textual language, without any iconic or spatial clues to help the querying process. A new generation of languages (visual languages) has been recently investigated, that attempts to make extensive use of the person’s instincts and senses. In this paper we propose a fully visual system, called Query by Diagram* (QBD*), which is based on a conceptual data model, a query language defined on this model and a graphical user interface. The main characteristics of the interface are the ease of use, and the availability of a rich set of primitives for both schema selection and query formulation. Unlike many present proposals of graphical query systems, graphical operations are formally defined from both a syntactic and a semantic point of view.

1. Introduction
A crucial point for the effective success of databases in organizations is the achieving of a friendly interaction of the user with the system. Experience suggests that traditional query languages [11], [27], [20] are not friendly enough for the casual user: she/he is requested to formulate queries in a textual language, without any iconic or spatial clues to help the querying process. Moreover, no mechanism is generally available in order to guide the user in the understanding of the database information content. A new generation of Query Languages (QL) has been recently investigated, that attempts to make extensive use of the person’s instincts and senses. Recent approaches in the literature can be classified as follows:

a) "intelligent" QL systems (25), [16], [18], [22]) which use artificial intelligence techniques to make man-machine interaction more similar to communication among people;
b) "graphical" QL systems ([13], [24], [26], [12], [10], [17], [7]) which use appropriate icons to represent objects, color, highlighting, and the natural "arrangement" of the objects.

The system presented in this paper, called QBD* (Query by Diagram*), is located in the second class. QBD* is characterized by:
a) a uniform graphical interface for both schema specification and query formulation;
b) the use of an Entity-Relationship [9] Oriented Data Model;
c) the availability of a relationally complete query language, whose expressive power allows also recursive queries.

The main idea of the system is to provide the user with a large set of Graphical Primitives, in order to friendly extract the required information from the database schema. Such an idea stems from the observation that several difficulties may arise in dealing with previously unknown database schemas, such as: "underground" implicit information, analyze too much detailed schemes, need of restore pre-acquired information, etc. The available graphical primitives are formally described by means of a uniform mechanism:

1) the database is defined in terms of a pair \( <G, M> \), where \( G \) is a graph corresponding to the intentional level of the representation, and \( M \) (called interpretation) is a set-theoretical structure corresponding to the extensional level;
b) the graphical primitives are functions mapping the database into a structure called the schema of interest, which is the subset of the database containing the concepts needed in the query formulation. The primitives are grouped into three classes, according to the type of objects they operate on: the graph (Location Primitives), the graph and the interpretation (Replacing Primitives), the interpretation (Navigational Primitives).

Our approach allows the user to deal uniformly with the same graphical environment during all the phases of the interaction with the database (without textual intermediate). The data model is very expressive and independent from the implementation model. Moreover, the set of the expressive graphical queries includes the class of recursive queries. In [3] we show this property by defining a textual language isomorphic to the graphical one, and by comparing the graphical operators with both the relational algebra and the class of computable queries (as defined in [7]).

The paper is organized as follows. In Section 2 we define the general architecture of the system. In Section 3 the definition of the data model is provided, together with some preliminaries needed for formally defining the graphical primitives, that are discussed in Section 4. In Section 5 we review the Query Language, extensively described in previous works [3], [6]. Finally, some conclusions are drawn in Section 6.

2. System architecture
In this section we present a general overview of the system, shown in fig 1: we assume that the DBMS is based on the relational model, although this is not a strict requirement.

Fig.1: Architecture of the query environment
The general architecture of the system is based on three main modules:
1. The Graphical Interface enables the user to query the database. The query is expressed on an E-R schema, and is built using the graphical commands available in the interface.
2. The Translator analyzes the queries and, using the target (relational) schema, translates the non-recursive queries into relational algebra expressions, and the recursive ones into programs in a host language.
3. The DBMS Interface translates the algebraic expression into a query formulated in terms of the target DBMS.

During the query formulation activity, the user is provided with a large set of facilities. In fact, she/he may first interact with the conceptual schema to understand its information content by means of a top-down browsing mechanism; then, using the location primitives, she/he may extract the subschema containing the concepts involved in the query (schema of interest). During this phase, the user may perform transformations on the schema of interest, by means of replacing primitives, bringing it "close to the query"; in this way, a (temporary or permanent) user view is built, that may be helpful in the subsequent activities of query formulation. Finally, the user may resort to the navigational primitives in order to complete the specification of the query, defining all its procedural characteristics. A complex query may be expressed by means of a sequence of elementary commands available in the interface.

In synthesis, the activity of query formulation is composed of several phases, each one supported by a wide set of primitives (location, replacing, navigational). In the sequel, we focus on the location and replacing primitives, while navigational primitives, which will be mentioned in the sequel, are used to denote the subschema of S selected by the user with the system.

In this section we give the definitions that will be used in section 4. In order to formally describe the syntax and semantics of the Graphical Primitives, let us state the following definitions:

Definition (typed graph)
A typed graph G is defined as a quadruple, <N(G), E(G), att, id>, where:
N(G) is the set of nodes; E(G) is the set of edges, i.e. E(G) ⊆ N(G) × N(G); att : N(G) → 2Ω a domain associating a domain to each attribute name if: N(G) → 2Ω such that id(n) ⊆ att(n), for all n ∈ N(G).

If n is a node, then we say that att(n) is the type of n.

If α ∈ Ω, then:
- n satisfies α iff α ∈ att(n).
- n satisfies α ∧ β iff n satisfies α and n satisfies β.

Definition (Database Schema)
A Database Schema S is defined by a pair:
S = <GS, MS>,
where GS is a typed graph and MS is an interpretation of GS.

Moreover, GS and MS verify several properties. In particular, we are interested in the following properties:
1) N(GS) = EntS ∪ RelS ∪ GenS, with EntS ≠ ∅, where:
EntS is the set of entity nodes; RelS is the set of relationship nodes; GenS is the set of generalization nodes (subset and generalization hierarchy);
2) E(GS) ∩ EntS ∩ EntS = ∅;
3) ∀ e ∈ EntS, ∀ x, y ∈ MS x ≠ y iff ν e[i] = id(e) (x[i] ≠ y[i]).
4) ∀ r ∈ RelS, ∀ q1, q2 ∈ EntS such that (q1,r,q2) ∈ E(GS), it holds that id(q1) ≠ id(q2).
5) ∀ g ∈ GenS, ∀ e1, e2 ∈ EntS such that (e1,g,e2) ∈ E(GS) 3 qj ∈ e1,gj such that ∀ i (Mg2 ∩ Mq1 ∧ id(qj) ≠ id(qj)). We call e1 the father in g.

Definition (Function of Interest)
We call Function of Interest any boolean function ranging on the set of nodes N(GS) of a database.

δ : N(GS) → {0,1}
The functions of interest represent the semantics of the selection operations performed by the user. Let us introduce three basic functions of interest.
1) If n is a node in the typed graph, then:
    δn (x) = 1 if x = n
2) If m, n are two nodes in the typed graph, then:
    δmn (x) = 1 iff x ∈ minpath (m,n)
where minpath stands for a function computing the minimal path between two nodes in a graph (see (11)).
3) If α is a boolean expression defined on a set of attribute names, then:
    δα (x) = 1 if x satisfies α.

The above functions correspond respectively to three selection cases: picking up a concept in the schema (the node n in the typed graph); selecting two concepts and asking for the minimal connecting path; specifying a boolean expression α defined on a set of attribute names. Let δ be a function of interest defined on a database schema S = <GS, MS>, we get:

Nδ (GS) = (l ∈ {0,1})
Eδ (GS) = (m,n) ∈ Eδ (GS) 1 ∧ δ(m) = 1
[Mδ]d1 = [Mδ]d2

The above definitions are used to denote the subschema of S selected by δ.

In this section, after discussing the browsing mechanism, we give both the informal semantics of each graphical primitive, and the formal syntax and semantics. We also present an example illustrating the interaction of the user with the system.

The example refers to a schema containing information about a university, and in particular about travels of professors and students. We suppose the user is interested in the following query: 'Find students taking part in a journey to Boston, which did not book any means of transport.'

4.1 Top-down Browsing Mechanism
The first activity a user performs when interacting with a schema that she/he did not know previously, is understanding its information content. To simplify this activity, the system provides a library of top-down refinements for each schema, documenting the design at various levels of detail, and supports a top-down browsing mechanism. Top-down browsing may be used selecting specific objects or whole schemas, and navigating up and down among the different top-down levels of the schema in order to better locate the fundamental concepts and the links among them. It is worth noting, that the top-down browsing may be very helpful also when used in connection with the location techniques described in section 4.2.1; in this case the user may locate the desired concept in an intermediate version of the schema: such selection results into the automatic selection of all the concepts of the final schema derived from it.

We show in fig. 3 two intermediate levels of the schema in fig. 2; the top-down development of the concepts of the schemas is displayed by means of different filling.

4.2 Graphical Primitives

The location and replacing primitives are two types of graphical primitives, defined as follows:

**Definition (Graphical Primitive)**

A graphical primitive (GP) \( p \) is a function from databases to databases. If \( S \) is a database \( p(S) \) is called *Database Schema of Interest*.

Two basic classes of GPs are discussed in this section: *Location primitives* and *Replacing primitives*. The former are used to locate a collection of nodes on the basis of a given function of interest; they do not alter neither the type of the nodes nor their interpretation; the latter replace a part of the database with a single node, which may have any type and interpretation.

In the following, we assume the GPs applied to a database \( S = \langle G_S, M_S \rangle \).

#### 4.2.1 Location Primitives

Location primitives are formally defined in the following way. Let \( S \) and \( f \) be a database and a function of interest respectively.

Two types of location primitives are defined:

**Definition (Simple Location Primitive)** A simple location primitive, w.r.t. \( f \), denoted as \( \text{loc}_f \), is defined as follows:

\[
\text{loc}_f(S) = \langle \langle G_S \rangle, M_S \rangle_f
\]

**Definition (Enlarged Location Primitive)** An enlarged location primitive, w.r.t. \( f \), denoted as \( \text{eloc}_f \), is defined as follows:

\[
\text{eloc}_f(S) = \langle \langle G_S \rangle, M_S \rangle_f
\]

Note that the Database Schema of Interest only depends on the function of interest without any reference to the type, and the interpretation of nodes. In other words, location primitives allow the incremental building of the schema of interest. For this purpose, the user may adopt three different strategies:

1) **Direct extraction** - The most immediate method of extracting the schema of interest, is to pick up the objects of the schema.
2) **Expression of meta-queries on the schema** - This strategy corresponds to querying the schema structure, and extracting from it meaningful information. For this purpose three methods are available:
   a) **Extraction by attributes** - The user may select the concepts by putting some conditions on their properties.
   b) **Extraction by path search** - The system provides an algorithm which searches all the existing paths connecting the two concepts. The user may specify conditions on the length of the path and/or on the presence of particular concepts. Such paths are displayed to the user which decides whether include them in the schema of interest.
   c) **Extraction using an "oil-stain" strategy** - Once a symbol, or a group of connected symbols, has been selected, it is possible to enlarge the sub-schema of interest by an "oil-stain" expansion.
3) **Use of a library of schemas** - The system is able to manage libraries of tailored schemas: schemas (or selected parts) can be extracted and integrated with the current schema of interest, using a "cut-and-paste" strategy. Analogously, the user may add new schemas to the library.

All the described activities add concepts to the schema of interest: since uninteresting concepts could be included in the result, the system provides the user with a de-location mechanism in order to eliminate such concepts.

We assume that in the previous phase of top-down browsing, the information content of the database has been fully comprehended by the user. So, the problem is now to extract the schema of interest needed in the query formulation.

The user may apply the location primitives in the following way:

- the desired top-down refinement of the schema is selected (level 2 in the example);
- the system shows a set of icons, each one representing a possible location strategy:
  a) the user points at the desired icon, in the example "Extraction by path search". At this point, the starting and the ending concept (PERSON and CITY) are picked up, moreover, TRIP is requested to appear in the path. The schema resulting from the above operations is shown in fig. 4. Fig. 5 represents the schema of interest at the most detailed level;
  b) the user de-locates the uninteresting concepts (PROFESSOR and TEACHES), and selects, by means of the direct extraction strategy, the
entities RESERVATION and TICKET, and the relationships LIVES and IN, producing the schema in fig. 6.

We may verify the correspondence between the graphical operations and the functions of interest by means of the above example (in the following we denote with $RS = \{GRS, MRs\}$ the schema in fig 2). We assume:
- $N(\text{GRS})$ is the set of nodes labeled with uppercase words;
- $\text{PERSON}, \text{TRIP}, \text{CITY}$ equal to the functions of interest involved in the “Extraction by path-search” in fig. 5 (note that the minimal path computed on the schema at the second refinement level, in fig. 4, automatically results, at the most detailed level, in the set of paths in fig. 5);
- $\text{RS}, \text{TICKET}, \text{LIVES}$ equal to the functions of interest involved in the “Direct extraction” in fig. 6.

At this point, the Database Schema of Interest is completely specified. Formally, it is defined as the composition of all the location primitives used:

$$\text{loc}_{\text{PERSON}} \cdot \text{loc}_{\text{TRIP}} \cdot \text{loc}_{\text{CITY}} \cdot \text{loc}_{\text{RS}} \cdot \text{loc}_{\text{TICKET}} \cdot \text{loc}_{\text{LIVES}}$$

(\text{RS})

(where $*$ is the composition operator).

4.2.2 Replacing Primitives. Once the schema of interest has been defined, the user is enabled to transform the schema itself into a new version, closer to the query expression. In this phase several replacing primitives are available.

The main difference between replacing and location primitives is that the former allow for modifying the schema, possibly with loss of information content. In other words, the user may build a proper view of the schema, which is not isomorphic to any schema of interest resulting from applying location primitives.

Replacing primitives may be reversible or non-reversible. Reversible primitives preserve the information content of the extensional level of the database. For example, if a generalization hierarchy exists among entities PERSON, MAN and WOMAN, the user may want to simplify the schema by deleting the entity PERSON. Such an operation is reversible, because the extension of PERSON can be obtained by the union of MAN and WOMAN.

On the other hand, the user may want to delete uninteresting information, producing a more concise schema. Operations of this kind are non-reversible.

A distinction must be made between primitives that operate on generalization hierarchies and other primitives. Inside a replacement area, primitives of the second type can be applied only when all generalization hierarchies have been eliminated. This choice is due to the need of formally defining the meaning of the replacing primitives, in terms of the semantics of the database schema.

Therefore, we first give the semantics of the primitive concerning the replacing of generalization hierarchies.

In order to specify the replacing primitives, we introduce the concept of $\text{Star}(n)$, where $n$ is a node. $\text{Star}(n)$ is the set of nodes at distance 1 from $n$. Let $g$ be a node in $\text{Gens}$, and let $e_1, \ldots, e_n \in \text{Star}(g)$ be the father in $g$. Several strategies may be adopted for replacing the node $g$. We describe here the two extreme cases (other cases may be easily derived).

G1) $g$ and the entity nodes $e_1, \ldots, e_n$ are replaced with an entity node $c_g$ such that:

$$\text{att}(c_g) = \text{att}(e_1) \cup \ldots \cup \text{att}(e_n)$$

$$|\text{M}_{c_g}| = |\text{M}_{e_1}| = \ldots = |\text{M}_{e_n}|$$

$$\forall x \in \text{M}_{c_g}, \exists e_i, \exists y \in \text{M}_{e_i} \text{ such that }\forall a, a \in \text{att}(e_k) \wedge x[a] = y[a] \iff x[a_{id(c_g)}] = y[a_{id(e_i)}]$$

G2) $g$ is replaced with $n$ relationship nodes $r_1, \ldots, r_n$ such that:

$$\forall e_1, \ldots, e_n \in \text{Star}(g) \exists (e_1, r_1), (e_2, r_2), \ldots, (e_n, r_n) \in \text{El}(g)$$

Besides the replacing of hierarchies, other primitives may be applied in any order to entities and relationships, their essential characteristic is the fact that the Database Schema of Interest is obtained by replacing a portion of the schema by a single node. Such primitives are distinguished into monadic and polyadic:

1. Monadic: only one node of $\text{GS}$ is replaced,
2. Polyadic: a single node $n$ together with $\text{Star}(n)$ are replaced.
The monadic primitives allow adding or deleting an attribute and replacing one schema concept with another. Attributes can be added only when they refer to the information content of the schema; so, they are to be defined by means of a generic query on the schema itself.

Concepts are replaced by other concepts specifying restriction conditions on their attributes. In this case, two types of monadic primitives are provided:

M1) a node n is replaced with a node n' such that:
\[ \text{att}(n') = \text{att}(n) \] and \( M_n \subseteq M_n' \)

M2) a node n is replaced with a node n' such that:
\[ \text{att}(n') = \text{att}(n) \] and \( M_n' = M_n \)

The polyadic primitives replace (with or without loss of information content) one central concept (n) linked to a generic number (m) of other neighbour concepts (\( n_t(n) \)), with a unique concept of the same kind of the neighbour concepts. Polyadic primitives fall under two categories:

P1) an entity node e and the relationship nodes \( r_1..r_n = \text{Star}(e) \) are replaced with a relationship node \( r_t \), such that:
\[
\begin{align*}
& \forall x \in M_e, \exists y \in M_r, \exists z \in M_{r_t}, \text{ such that:} \\
& \text{att}(e) = \text{att}(r_1) \cup \ldots \cup \text{att}(e_t) \\
& \{x[\text{att}(e_i)] = y[\text{att}(r_j)], \ldots, y[\text{att}(r_k)] = z[\text{att}(r_t)]\} \\
& \forall x \in M_e, \exists y \in M_{r_t}, \exists z \in M_{r_t}, \text{ such that:} \\
& \{x[\text{att}(e_i)] = y[\text{att}(r_j)], \ldots, y[\text{att}(r_k)] = z[\text{att}(r_t)]\}
\end{align*}
\]

P2) a relationship node r and the entity nodes \( e_1..e_n = \text{Star}(r) \) are replaced with an entity node e, such that:
\[
\begin{align*}
& \forall x \in M_r, \exists y \in M_{e_1}, \exists z \in M_{e_t}, \text{ such that:} \\
& \{x[\text{att}(r)] = y[\text{att}(e)], \ldots, y[\text{att}(e_t)] = z[\text{att}(e_t)]\} \\
& \forall x \in M_r, \exists y \in M_{e_t}, \exists z \in M_{e_t}, \text{ such that:} \\
& \{x[\text{att}(r)] = y[\text{att}(e)], \ldots, y[\text{att}(e_t)] = z[\text{att}(e_t)]\}
\end{align*}
\]

We now describe the interaction of the user with the system, in particular referring to the schema in fig. 6:

a) the user points at the desired icon:

- Monadic case: the system first enables the user to select the concept to be replaced in the schema. Thereafter a window is shown, displaying the attributes of the concept. The "add attribute" case results in specifying the corresponding query. In the example, the user derives the attributes STARTING_CITY and ENDING_CITY for the entity TRIP, by means of the paths TRIP-TO-CITY, and TRIP-FROM-CITY. Similarly, the attributes LIVING_CITY for the entity PERSON, and TO_CITY for the entity TRAVEL are defined;

- Polyadic case: the user specifies the central concept in the part of the schema to be replaced, then the system focuses on the concepts involved in the transformation. In the example, the user collapses the hierarchy RESERVATION-TYPE-TICKET in the entity TICKET, and the hierarchy PERSON-KIND-STUDENT in the lower entity STUDENT.

The result of all previous replacing activities is shown in fig. 7.

In this section the formal semantics has been provided only for the basic replacing primitives. All the other replacing primitives, informally described, may be derived from the composition of the basic ones. For example, adding the attribute LIVING_CITY to the entity PERSON may be realized by means of two steps: 1) the polyadic primitive P2 applied to the path PERSON-LIVES-CITY; 2) the monadic primitive M2 applied to the resulting entity, deleting all the attributes of CITY apart from CITY_NAME (renamed in LIVING_CITY).

Generally, all the non-reversible primitives may be obtained by applying the primitives M1 and M2 to the result of the application of the other reversible primitives.

5. Navigational Primitives

After the schema of interest building, the user may work on the interpretation by using the navigational primitives. Such navigational primitives constitute a graphical query language whose expressive power allows recursion to be expressed. In this section we review such query language. The language is a visual navigation language on E-R schemes, represented by means of diagrams; the basic idea is to decompose a query into its elementary steps, expressed by a limited set of graphical operations, such as choice of icons, selection of concepts, and navigation. In order to formally express the syntax and the semantics of the language, a one-to-one correspondence between the graphical operations and the syntactic constructs of a textual query language is defined.

The general structure of the query is based on the location of a distinguished concept, called main concept (entity or relationship), that can be seen as the entry point of one or more subqueries. Subqueries can be combined by means of the usual union and intersection operators. Such subqueries express possible navigations from the main concept to other concepts in the schema, in order to extract information from any number of different entities or relationships. Once the main concept has been chosen, two different types of primitives are available for selecting its instances. The first one (called FIND in the textual language) is based on the idea of following paths of linked concepts and establishing conditions on them; the second one (called BRIDGE) is used for joining two concepts without following schema links, just comparing their attributes. Such two primitives can be arbitrarily nested.

In the following, we give the informal semantics of each graphical primitive, showing the interaction of the user with the system in the navigational phase, and referring to fig. 7. First, the user selects all the students taking part in the journey in Boston; after that, she/he extracts from the above set of students those which have completed the reservation of tickets. To find such students it is necessary first to select all booked trips, then to compute the transitive closure of the result, looking for the chains having the attribute FROM_CITY equal to the attribute LIVING_CITY, and the attribute TO_CITY equal to "Boston". The difference between the two previous student sets gives the desired result, i.e. the set of students that have not completed the reservations.

The general sequence of the user operations is the following:

- selecting the icon corresponding to "navigation";
- choosing the main concept of the query, by picking it up with the mouse (the entity STUDENT in the example);
- extracting the desired instances from the main concept, specifying possible conditions on its attributes. The list of attributes is shown in a separate window, containing also the elements involved in the
comparison (i.e., constants, other attributes, etc.), and a set of icons suitable to formulate conditions on the attributes. Conditions are expressed selecting the attributes and the icon corresponding to the required operator. The system shows the result of the operation by displaying a graph, where vertices correspond to attributes, and labelled edges correspond to the operators;

- extracting the desired instances from the main concept, specifying conditions on different concepts. This operation may be carried out either by picking up an existing path of concepts, or joining concepts, by comparing their attributes using the windowing mechanism described above.

It is worth noting that QBD* allows performing non-first order queries by means of a simple graphical mechanism, which is the same underlying the BRIDGE operator. In fact, an operator computing the generalized transitive closure (GTC) [19] is available (the GTC is a transitive closure of an entity, in which the cycle conditions are extended to be boolean expressions with equality and inequality operators). Such cycle conditions are equivalent to particular BRIDGE conditions where an entity is compared with itself, therefore the window used for the GTC looks like the one adopted in the BRIDGE definition, the difference being the pre-selection of a particular icon, that signals the beginning of a recursive session. In such a session the user may:

- select the desired entity;
- specify the cycle conditions on the attributes of the entity. In the example, the cycle condition is the equality of the attributes FROM_CITY and TO_CITY;
- specify a set of conditions in order to select a subset of the result of the transitive closure.

The main idea is to visualize the generic step of recursion (corresponding to a cartesian product) displaying a double list of the same attributes; the user may specify the cycle conditions comparing the attributes with each other. Then, the system determines the attributes involved in the projection.

During the query formulation the following features are also available:

- building new attributes either by manipulating existing attributes, or by renaming them. For this purpose a set of functions typically supported by advanced query languages, such as SQL [23], is available (COUNT, SUM, AVG, etc.), in addition to the mathematical operators;
- storing queries, or parts of them, in a query library. Each query is stored together with a label, automatically generated during the query formulation. Such a label, expressed in a restricted natural language, is used as an echo, to check the correctness of the query generated by the system;
- selecting the icon corresponding to the set operators (union, difference, intersection), for composing different subqueries.

6. Conclusions

We have presented the query system QBD*, that allows the user to interact with a database in a fully graphical way. Moreover, we have defined an Entity-Relationship Oriented Data Model within the context of a more general graph-based model. On the basis of such a model, we have formally described the syntax and the semantics of a set of graphical primitives, which have been implemented in the system. It is worth noting that the formalization mechanism is not strictly dependent on the particular graphical primitives of QBD*. Rather, it may be adopted in order to describe graphical primitives working on a generic diagrammatic representation.

A prototype of the QBD* has been developed at Dipartimento di Informatica e Sistemistica of the University of Rome "La Sapienza": the C-language is the development environment, operating system MS-DOS; the graphical interface is supported by the tool PAINTER, part of the MAST-ER Case tool by GESI s.r.l. [15], using algorithms for automatic layout described in [21].

Further research will be devoted to the extension of the system for dealing with statistical data. In this case, the query must allow the definition of aggregation and statistical operations; the graphical interface is based on the model defined in [5].

Acknowledgements

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

REFERENCES