Journal of Visual Languages and Computing (1996) **7**, 267–289

Implementing Hyperlog, a Graph-Based Database Language

STEFAN HILD AND ALEXANDRA POULOVASSILIS

Department of Computer Science, King's College London, Strand, London, WC2R 2LS, U.K.

Received 13 July 1994 and accepted 28 August 1995

We describe the implementation of Hyperlog, a graph-based declarative language which supports both queries and updates over a graph-based data model called the Hypernode Model. This model is capable of representing arbitrarily complex data structures by means of nested and recursively defined graphs, while the Hyperlog language is computationally complete. By requiring only a very small number of graphical constructs, Hyperlog is well-suited to non-expert database users. In this paper we describe the graphical aspects of the Hyperlog implementation including novel techniques for: representing and updating data, queries and programs; representing and browsing the database; and representing and browsing the output from queries.

1. Introduction

MUCH RECENT database research has focussed on deductive [4] and object-oriented [12] databases, and also on graph-based representation and manipulation of data. Following the general direction of these trends, a graph-based data model called the Hypernode Model has been developed [14, 15] which exhibits features of all three. The model is based on nested, and possibly recursively defined, graphs termed *hypernodes.* Hypernodes have unique labels which serve as object identifiers: this means that arbitrarily complex objects can be represented by a network of hypernodes. Hypernodes can contain the labels of other hypernodes in their node-set, including their own label: this means that the model provides inherent support for the nesting of information.

The Hypernode Model comes equipped with a graph-based declarative language called Hyperlog which supports both queries and updates. A query in Hyperlog consists of a number of graphs (termed *templates*) which are matched against the hypernodes in the database and which generate graphical output. Database updates in Hyperlog are undertaken by means of programs. A Hyperlog program consists of a set of rules whose head and body are again sets of templates. The templates in the head of a rule indicate the updates to be undertaken for each match of the templates in the body of the rule. The evaluation of a program consists of a repeated matching of the bodies of its rules against the current database state until no more updates can be inferred.

1045-926X/96/030267 + 23 $18.00/0

A discussion of the rationale behind the design of Hyperlog, including its semantics and expressiveness can be found in [19]. Hyperlog was designed for non-expert database users. Thus, as we will see below, a very small number of graphical constructs are used to express queries and updates of arbitrary complexity. This paper is concerned with the implementation of the language, concentrating in particular on its graphical aspects. Key issues addressed include the following:

(i) Graphical techniques for representing and updating data, queries and programs: in particular, we will see how all of these are created by instantiating currently available types, thereby guaranteeing their type-correctness.
(ii) Graphical techniques for representing and browsing the database: we will describe a novel algorithm for the automatic display of the contents of a database which uses concepts from mechanics and self-organising networks.
(iii) Graphical techniques for representing and browsing the output from queries, which can be arbitrarily large: we regard the query output as data which can be stored and browsed in the same way as the database.

We begin the paper with a brief introduction to Hyperlog in Section 2, which sets the scene for the rest of the paper. In Section 3 we describe the architecture of our implementation. Section 4 is the key section of the paper and is concerned with the graphical aspects of the implementation, covering points (i)–(iii) above. Section 5 compares Hyperlog with other graph-based database languages and highlights some of its distinguishing features. Finally, Section 6 gives our concluding remarks and directions of further work.

2. Overview of Hyperlog

2.1. Hypernodes, Types and Instances

A *hypernode* is a triple (c, N, E) where c is called the *label* of the hypernode and (N, E) is a directed graph. A *hypernode repository* (or simply a *repository*) is a set of hypernodes no two of which have the same label.

We distinguish two relationships between the hypernodes in a repository: *links*, which represent the containment of one hypernode within the node-set of another, and *edges,* which connect two nodes in the node-set of the same hypernode. As we will see below, both these relationships are displayed to the user graphically.

Hyperlog is a typed language, where both *types* and *instances* are represented by hypernodes. In addition to such user-defined graph types, the language also supports primitive types such as STRING, NUMBER and TEXTs. Poulovassilis and Levene [19] discusses the representational expressiveness of hypernodes. In particular, we show there how hypernodes can be used to represent conventional data structures such as mappings, tuples, records and sets, to an arbitrary level of nesting. The Hypernode Model can thus be used to visualise a wide variety of data models, including functional, relational and object-orientated ones.

In this paper, we concentrate mainly on object-based modelling. We use as our

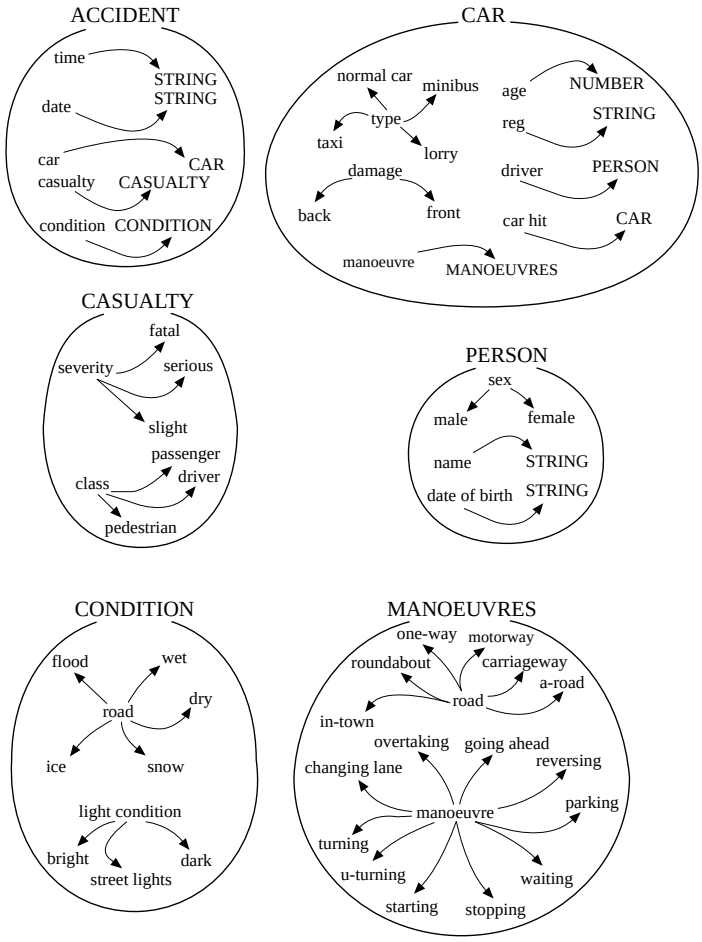

Figure 1. The Road Accident Database scheme

running example a database that records Road Accident Data. Figure 1 shows the schema of this database (where a database *schema* is just a set of types). Upper-case identifiers within types represent further types (either graph types or primitive types) whereas lower-case identifiers represent user-defined constants: these appear within instances as they stand and correspond to user-defined enumerated types in programming languages. Thus, the type ACCIDENT in Figure 1 shows that each accident instance contains information about the time, date, cars, casualties, location and condition of the accident, where CAR, CASUALTY and CONDITION are further graph types. The type CAR contains some user-defined constants: a set of valid car types and a set of types of damage. It references the graph types PERSON, MANOUEVRES and CAR (recursively). The types CASUALTY, PERSON, CONDITION and MANOUEVRES are self explanatory.

Figure 2 illustrates some instances of the types of Figure 1. As we discuss in greater detail in Section 4.3, instances are created by replacing type-nodes by instance-nodes. For example, ACCIDENT_12 is an instance of ACCIDENT and, within it,

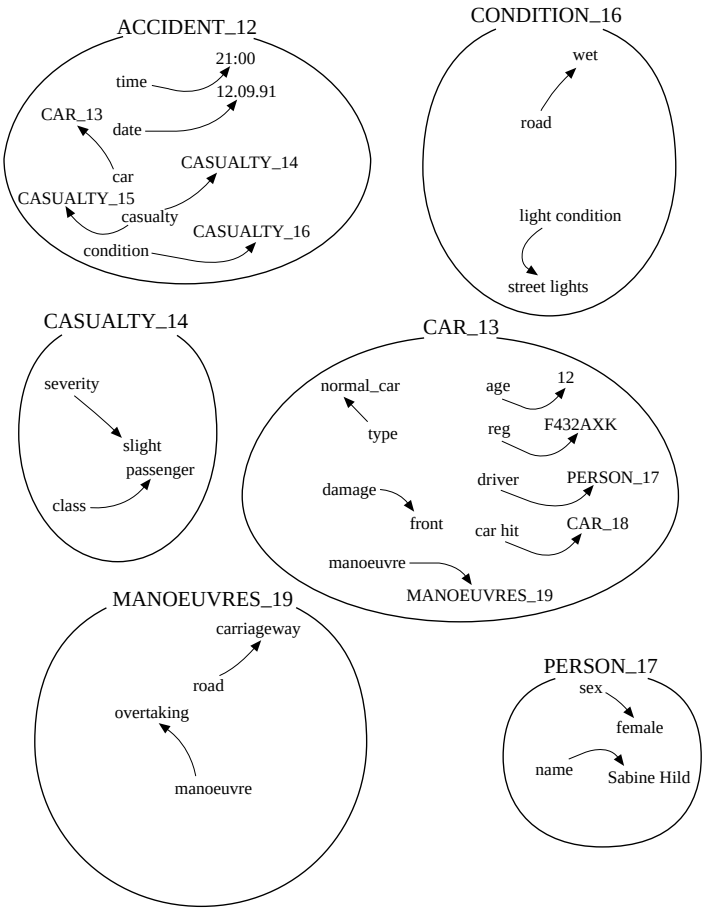

Figure 2. Part of the Road Accident Database

CAR_13 is an instance of CAR, CASUALTY_14 and CASUALTY_15 are instances of CASUALTY, and CONDITION_16 is an instance of CONDITION. Note that any number of instances of nodes and edges can be created, for example ACCIDENT_12 has two casualties.

2.2. Queries

A Hyperlog query is a set of templates. Templates differ from hypernodes because they can contain variables as their labels or nodes and because their nodes and edges can be negated. For example, Figure 3 shows a query that determines the time and date of accidents that were not caused by Alexandra P. In queries, nodes that are prefixed by '?' or '!' are variables for which matches need to be found with respect to the database. Variables marked '?' are ones for which the user desires their matches to be output whereas variables marked '!' are intermediate ones which are to be used for

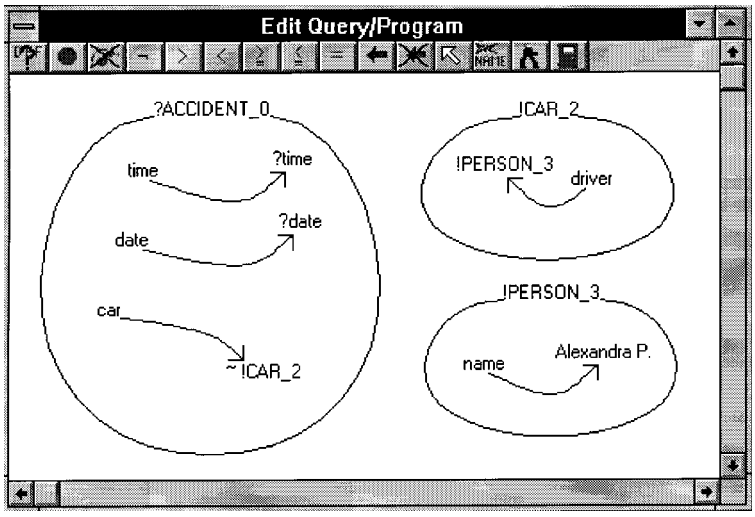

Figure 3. A query within the Query/Program Editor

matching purposes only. We discuss the handling of query output in greater detail in Section 4.4.

2.3. Programs

A Hyperlog program is a set of rules. Each rule is an expression of the form

$$h_1, h_2, \ldots, h_n \Leftarrow b_1, b_2, \ldots, b_m$$

where $m, n \geq 0$, and $h_1, \ldots, h_n, b_1, \ldots, b_m$ are templates. The part of the rule that appears to the left of the $\Leftarrow$ symbol is the *head* of the rule and the part to the right is the *body*.

The templates in the head of a rule indicate the updates to be undertaken for each match of the templates in the body of the rule. The evaluation of a program consists of repeatedly (i) matching the bodies of the rules against the current database state, and (ii) updating the database with the updates inferred. The evaluation terminates when no new updates can be inferred. To illustrate, Figure 4 shows a program to find the age of the oldest car involved in any accident and to place this result in a hypernode labelled RESULT. Notice that in programs all variables are used for matching only, irrespective of whether they are marked '?' or '!'.

This program comprises two rules and assumes that two hypernodes labelled RESULT and FLAG have already been created. The upper rule matches all instances of type CAR and places their ages into the RESULT hypernode. It also places a 'done' node into the FLAG hypernode, thus preventing itself from firing on subsequent rounds of the program execution. The lower rule fires only when there are two or more nodes in RESULT (so it does not fire on the first round of the program execution). It deletes all values ?Y in RESULT which are less than some value ?X in RESULT. Overall, if there are 0 or 1 distinct car ages, the program terminates after the second round with at most one age in RESULT. If there are more than 1 distinct

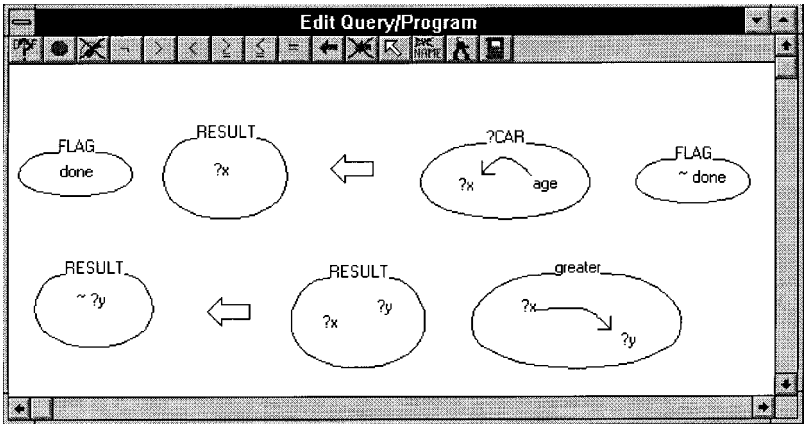

Figure 4. A program within the Query/Program Editor

car ages, the program terminates after the third round with the greatest car age in RESULT.

Notice the hypernode with label 'greater' in the body of the second rule in Figure 4: this is a graphical representation of the built-in operator >. An alternative way to represent this information would have been to create an edge between ?X and ?Y in RESULT and annotate it with the > symbol from the tool bar. In either case, the operator > is built-in only for reasons of efficiency since Hyperlog is computationally complete. For example, [19] discusses how arithmetic can be carried out in the language. Furthermore, Hyperlog is update-complete i.e. it can be used to perform any computable database transformation. This aspect of the language is also discussed in [19].

3. Implementation Architecture

Figure 5 illustrates the architecture of our Hyperlog implementation. We observe from this figure that three repositories are used: one for types, one for queries and

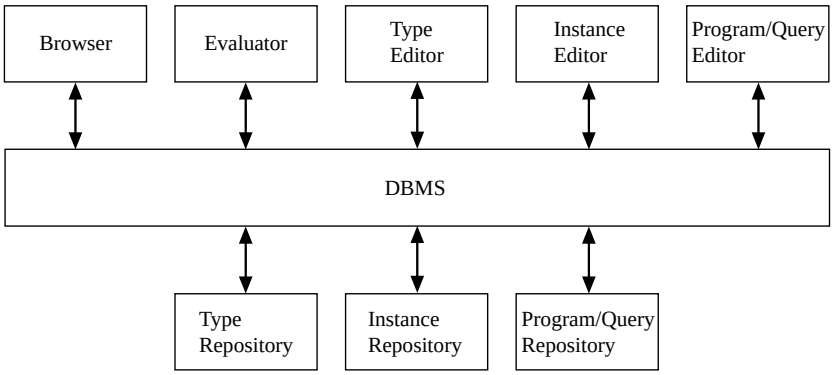

Figure 5. The implementation architecture

programs, and one for instances. The three corresponding Editors provide graphical facilities for the creation, viewing and update of this information: we describe these editors in Section 4.

The instance repository is further sub-divided into *domains,* which are numbered 0, 1, 2, etc. Domain 0 contains the data, i.e. the instances of the currently defined types. Domain 1 contains the meta data, i.e. it is a 'reflection' of the type repository, so that the schema can be queried and navigated (but not updated) in the same way as the data. As we will see below, a new domain is generated after the evaluation of each query in order to hold the query output. Such domains can subsequently be viewed by using the Browser and can be deleted at any time.

The Browser generates the containment graph of a selected domain of the instance repository and presents it to the user. In particular, the results of a query can be viewed in this way. As we will see in Section 4.5, various reduction strategies are provided for further restricting the instances to be displayed by the Browser. In addition, specific instances can be selected from the Browser display and their contents edited by the Instance Editor. A trail of the last 20 visited instances is automatically maintained within the Browser display.

The Evaluator reads a query/program from the query/program repository and evaluates it with respect to a selected domain of the instance repository. In the case of a query, the result of the query is stored within a new domain, ready for subsequent viewing by the Browser. In the case of a program, the selected domain is updated with the changes inferred. The Evaluator employs algorithms which in fact perform a much more efficient evaluation of programs than just repeatedly matching rule bodies with respect to the entire domain on each round: [Ben93] gives a preliminary account of the optimizations employed.

Notice that there is no strict 1–1 mapping between the repositories and the components of the architecture. The Evaluator, for example, accesses all three repositories to perform its task, while the Instance Editor accesses both the type repository and the instance repository. The Database Management System coordinates access to the repositories by the other components and takes care of buffering and caching of frequently-used information. We refer the interested reader to [22] for a discussion regarding the implementation of hypernode repositories, which we do not consider further in this paper.

4. Graphical Features of the Implementation

The graphical features of our implementation can be summarized as follows:

(i) the display and update of individual hypernodes within the Type Editor;
(ii) the display and update of individual hypernodes within the Instance Editor;
(iii) the display and update of a query or program within the Query/Program Editor;
(iv) the display of a domain within the Browser, including the node placement algorithm, the reduction strategies, and the database navigation aids.

In Section 4.1 we describe the functionality of the Type Editor while in Section 4.2

we discuss the issues involved in the automatic graphical display of hypernodes. In Sections 4.3, and 4.4 and 4.5 we address the Instance Editor, the Query/Program Editor and the Browser, respectively.

4.1. The Type Editor

A hypernode database schema consists of a set of types which are created and updated using the Type Editor. Figure 6 illustrates the user interface of the Type Editor, with the type CAR of Figure 1 being edited. The tool bar located beneath the title bar allows the user to select from a number of options namely, from left to right: delete the current type, create a constant-annotated node, create a type-annotated node, delete a node, create an edge, delete an edge, return to normal edit mode after any previous operation, rename a node, and exit.

All operations on a type are performed by means of mouse-pointing. Most operations require the user to select the object that is to be manipulated and then to choose one of the options from the icons in the tool-bar. Nodes and edges are selected by clicking close to them and the type itself is selected by clicking close to its name. Edges are drawn automatically by the system and nodes can be moved around within the type—if they are connected to an edge, this follows them around automatically.

Nodes or edges can be added to, or deleted from, a type at any time. This of course has implications on the current instances of that type. In particular, nodes or edges can be added to a type without affecting the type-correctness of its instances: these will simply record no information with respect to the new nodes or edges. However, deletion of a node or an edge from a type results in the deletion of the corresponding information from its instances (after confirmation by the user). Note that deletion of a node within a type or instance automatically causes the deletion also of any edges incident upon it.

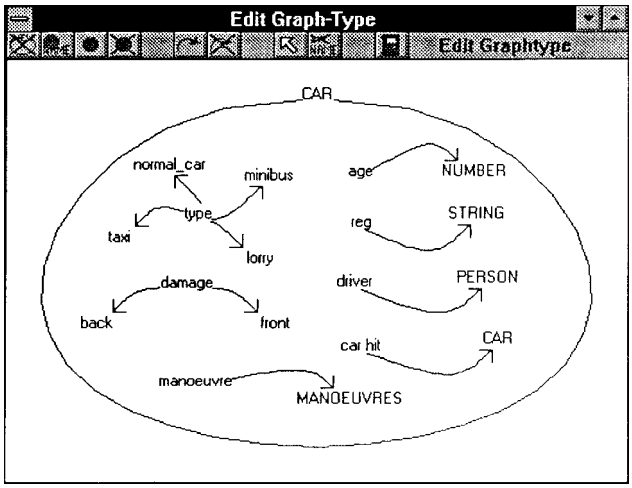

Figure 6. A type within the Type Editor

4.2. Drawing Hypernodes using Bezier Curves

From the description of the Type Editor above we observe that the user manually performs the placement of nodes within a type while the system automatically performs the drawing of the surrounding outline and of the edges between nodes. The problem we faced in supporting this functionality was to find a formula that could be easily and efficiently computed, and which generated smooth rather than angular graphics. A solution for the implementation of both the outline and the edges of a hypernode is to use a Bezier curve [21].

Given a set of vertices, the Bezier formula produces a smooth curve from the first vertex to the last vertex, with the intermediate vertices defining the derivatives, order, and shape of the curve. In general, the vertices form an open polygon, as shown in Figure 7. The Bezier curve will tend to follow the polygon shape, and so moving the vertices of the polygon gives fine control over the final shape of the curve.

The mathematical basis of the Bezier curve is a polynomial-blending function which interpolates between the first and the last vertices. The Bernstein polynomials are used as weighting functions for each vertex. These polynomials are given by the following formula, where n is the degree of the polynomial, i is the number of the particular vertex (the vertices being numbered from 0 to n) and $\binom{n}{i} = n!/i!(n-i)!$:

$$J_{n,i}(t) = \binom{n}{i} t^i (1-t)^{n-i}$$

The Bezier curve is then defined by a function $B(t)$ where $0 \leq t \leq 1$ and P_i is the position of vertex i:

$$B(t) = \sum_{i=0}^{n} P_i J_{n,i}(t)$$

Two useful properties of this function are as follows:

- $B(0) = P_0$ and $B(1) = P_n$ i.e. the curve is guaranteed to start at the first vertex and to end at the last vertex
- the rth derivative of the curve at its start and end is determined by the first r and the last r vertices respectively; in particular, the first derivative (i.e. the slope) of the

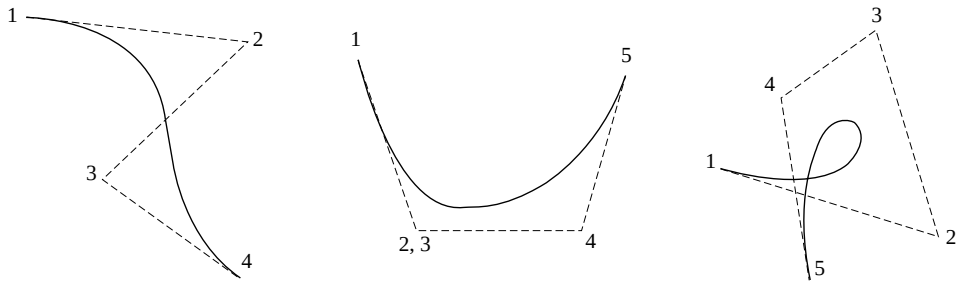

Figure 7. Bezier curves

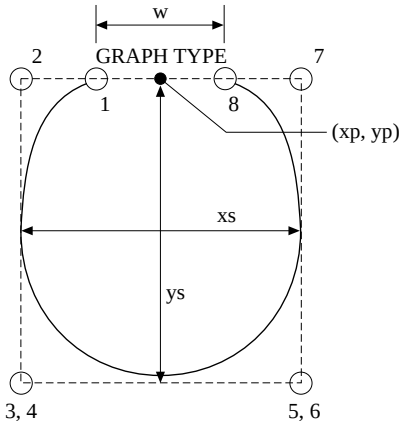

Figure 8. Drawing outlines

curve at its start and end is governed by the first two and the last two control points, respectively.

4.2.1. Drawing Outlines

Since we are using a Bezier curve, this problem reduces to identifying suitable control points to generate the outline. Experimentation has shown that 8 control points can be used to generate a satisfactory curve and Figure 8 shows the placement of these 8 points. The outline starts and ends at either side of the name of the type, whence the placement of control points 1 and 8. Control points 2 and 7 are placed at the same height as 1 and 8, and ensure that the shape starts and ends with a near-horizontal slope. The other two positions of interest are the bottom-right and bottom-left edge of the shape. Two control points are placed at each of these positions, giving them more weight in $B(t)$. This ensures that the outline expands towards these positions and that the hypernode has more room for its contents.

The only information required to reproduce the outline of any hypernode is thus its name, the position of the centre of the name, and the maximum width and height of the hypernode. This information is stored together with the node- and edge-set of the hypernode. Given this information, the coordinates of all 8 control points can easily be reproduced.

4.2.2. Drawing Edges

Edges are somewhat more demanding than outlines to draw. An edge can connect two different nodes or can connect a node with itself.

For the first of these cases the problem reduces to identifying suitable control points for a Bezier curve. The arrow at the end of an edge can be drawn very simply if the edge hits the target node at an angle of 45 degrees (since the two lines making up the arrow are then just horizontal and vertical lines, respectively). Since the Bezier curve is guaranteed to end tangentially to the line defined by the last two control points, this can easily be achieved by using a suitably-placed control point near the target node. Four cases for its placement can be distinguished, determined by the

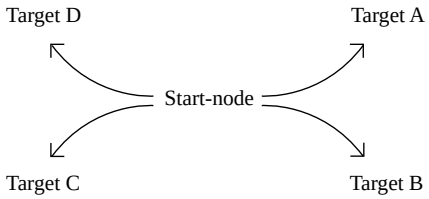

Figure 9. Relative positions of the end-node to the start-node

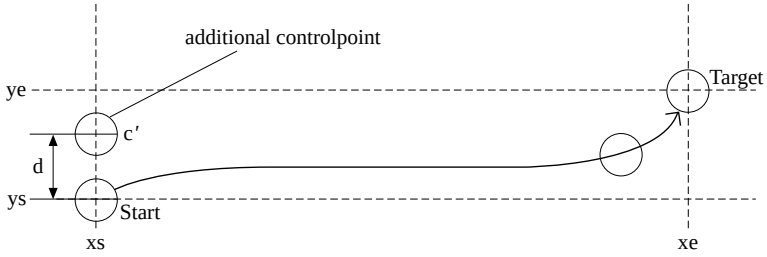

Figure 10. The four control points for an edge

relative position of start and target node to each other—see Figure 9. Thus, the control point will be placed in the appropriate quadrant at an angle of 45° to the target node, and at some experimentally determined distance.

The above arrangement will produce nice looking edges if the start node and end node are not too far apart. If they are, then the edge looses its elegant arch with the effect that it looks more like a straight line. Our algorithm checks for this situation and inserts another control point either above or below the start node. Figure 10 illustrates the look of such an edge, where the additional control point is set if $|xe - xs| \geq dx$ and $|ye - ys| \geq dy$, for some experimentally determined constants dx and dy.

The second kind of edge is one which starts and ends at the same node. There are no sub-cases to consider here and Figure 11 shows the placement of the control points required.

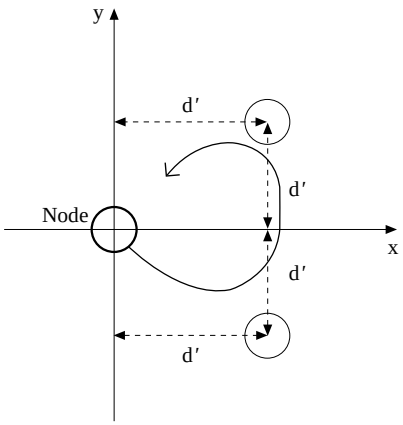

Figure 11. The three control points for a cyclic edge

4.3. The Instance Editor

Instances of types are created and updated using the Instance Editor. Since Hyperlog is strongly typed, a new instance is created by first selecting a type for it. A unique label for the instance will then be generated by the system, but the user is also given the option of changing this to some other unique label. For example, Figure 12 shows the display within the Instance Editor just after the user has chosen to create an instance of type PERSON and the label of the instance has been set to PERSON_32. At this stage, PERSON_32 has been inserted into the instance repository but as yet contains no nodes or edges. As we can see from Figure 12, the contents of the type PERSON are displayed within the new instance. To avoid confusion, this type information appears in a different colour (green) than the instance information (black). As with the Type Editor, the tool bar allows the user to select from a number of options namely, from left to right: return to the previous instance that was being edited, switch off the type information (see below), delete a node, delete an edge, return to normal edit mode after any previous operation, rename a node and exit.

The user can now instantiate nodes of the (green) type definition by double-clicking on them. When creating an instance of a primitive type such as a string or a number, the editor prompts the user to enter the value of this node and accepts it if it is type-correct. When creating an instance of a graph type, the editor creates a unique label but allows the user to override it with another new label or with the label of an existing hypernode of the correct type.

Edges can be created between any two nodes that are of the same type as the start and end nodes of an edge in the type-definition: as in the Type Editor, edges are drawn automatically. In addition, constants are inserted automatically as soon as an edge is created that originates or terminates there.

Any number of instantiations of nodes or edges can be made. Since these are type-checked, the type-correctness of the instance is guaranteed. The Instance Editor provides the same facilities for deleting and moving nodes and edges as the Type Editor.

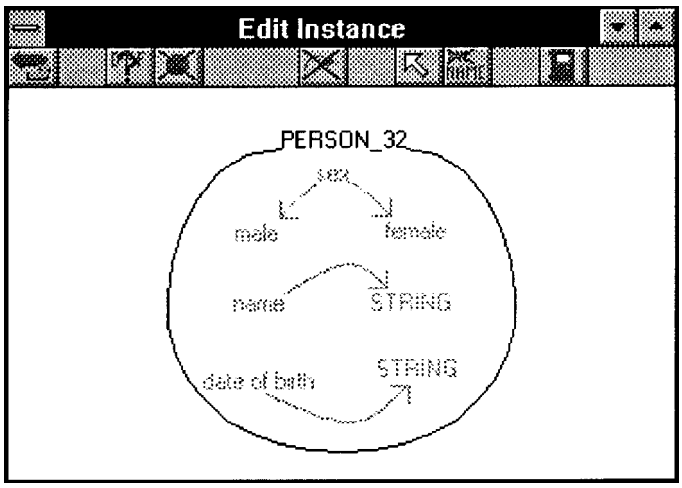

Figure 12. Creating an instance with the Instance Editor

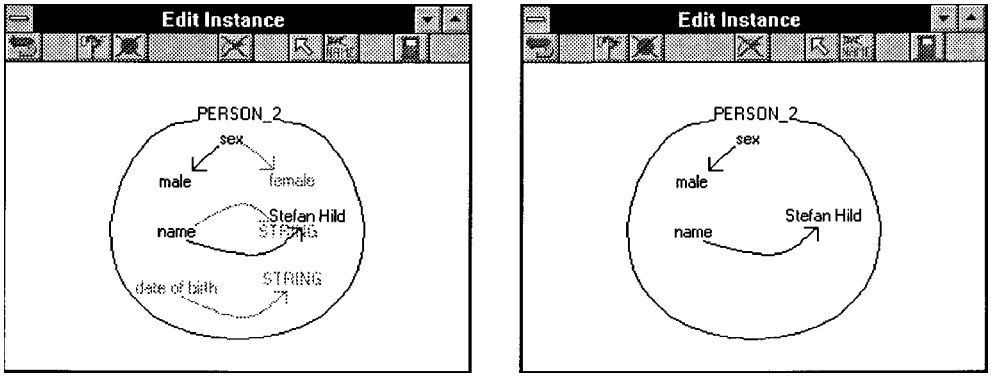

Figure 13. Viewing an instance with the Instance Editor

When viewing an instance, the super-imposed type definition may be distracting: see, for example, the left-hand screen of Figure 13 which illustrates an instance of type PERSON. The Instance Editor therefore provides a function which allows the user to switch off the type definition, and this is the default option when the Instance Editor is called on to view an instance that is already in existence (the right-hand screen of Figure 13).

After creating an instance, the user can move 'into' that instance by double-clicking on one of its nodes. If this node is a text file, a Text Editor is invoked to edit it. If the node is a label, the Instance Editor presents the contents of this next instance for viewing or update. This navigation process can be repeated as many times as desired. The trail of visited instances is displayed in the Browser window, in addition to the current Browser selection.

4.4. The Query/Program Editor

This editor provides all the functions necessary to create and manipulate the templates representing a query or program, which we consider in turn below.

4.4.1. Queries

Figure 3 above illustrates the editor display with a query currently being edited. The editor provides similar functionality to the Instance Editor in that it allows the creation of templates from types. However, its visual space is strictly 2-dimensional and so that all the query templates exist on one screen (which is vertically and horizontally scrollable). The tool bar options in Figure 3 (and Figure 4) are, from left-to-right: switch off the type information, create a node, delete a node or edge, add a negation annotation to a node or edge, add a comparison operator annotation to an edge (the next 5 icons), create an arrow symbol (only applicable for a program), delete an arrow symbol, return to normal edit mode, rename a node, execute a query or program, and exit.

Similarly to instances, queries are created by instantiating one or more types into one or more templates each. For example, the query in Figure 3 was created by instantiating the types ACCIDENT, CAR and PERSON. An important consequence of this functionality is that no searching that violates the database schema (and is therefore bound to find no matches) is ever undertaken.

While matching a query, the Evaluator has to be able to distinguish between variables that need to be matched, and identifiers which are constants. We adopted the convention that variables are marked by the prefix '?' or '!' and the Evaluator only generates output for those variables with prefix '?'.

A problem that we faced in supporting queries was how to handle their output in a flexible fashion. The solution we came up with was to generate hypernodes within a new domain from the matches for '?' variables. The Browser can then be used to view this output in an identical fashion to the way that the data in Domain 0 is viewed. The Evaluator thus applies each matching substitution that it finds to the '?' variables in the query and inserts the resulting hypernodes into a new domain, where they await selection for browsing by the user.

4.4.2. Programs

For the entering of programs, the Query/Program Editor also allows the placement of left-arrow symbols amongst the templates. Apart from this slight extension, the functionality provided for entering queries is sufficient to cater for programs too.

A program contains purely graphical data and so the Evaluator's first task upon being passed a program to evaluate is to identify 'clusters' in the program layout and thus generate the individual rules. By analysing the coordinates of each template, the Evaluator can assign it to a specific rule and also decide whether it is part of the head of the rule or part of the body. In order to assign templates to rules, the Evaluator makes use of the fact that each rule contains exactly one arrow, and that its templates stretch out horizontally to the right and left of that arrow. The number of rules in the program is assumed to be the same as the number of arrows placed by the user. The rules are constructed around these arrows by assigning each template to that rule whose arrow is vertically closest to the template. For example, Figure 14 shows the splitting of a set of templates into two separate rules. Since the templates in rules are

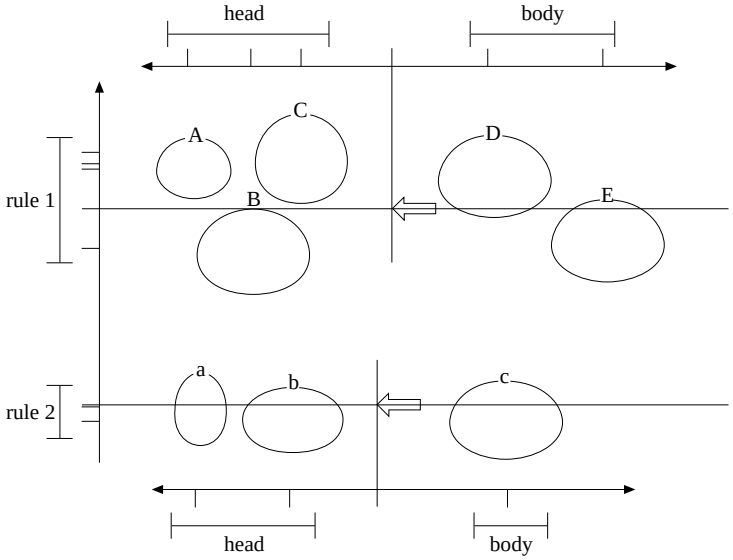

Figure 14. Splitting templates into rules

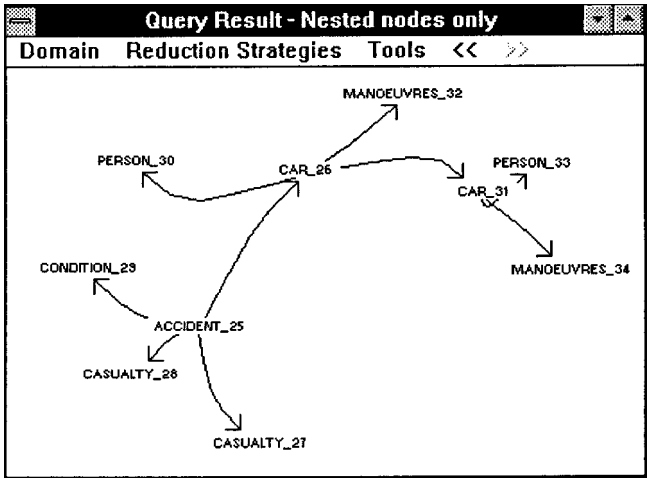

Figure 15. Display of a database domain

created from types, it is easily seen that a program does not undertake any type-incorrect (and therefore redundant) matching, and does not make any updates that will violate the type integrity of the repository.

4.5. The Browser

The Browser gives the user an overview of the structure of a selected domain of the instance repository. In particular, it displays the labels of hypernodes within a selected domain together with the containment relationships between them. Thus, if a and b are both hypernodes and b is in the node-set of a, the Browser will display a link $a \rightarrow b$. Figure 15 illustrates the display of a domain within the Road Accidents database. Note that primitive nodes such as numbers, strings and text are not displayed since they represent database contents rather than database structure. The Browser allows the user to access a particular instance by double-clicking on it. This will call up the Instance Editor for viewing or update of the instance. The Browser display is refreshed after each update within the Instance Editor since this may affect the containment relationships.

Since recursion and nesting are underlying principles of the Hypernode Model, the Browser is a particularly useful and indeed essential component of the system. A number of issues have been addressed in its implementation: first, a domain may be further broken down before being browsed by using one of a number of practically useful reduction strategies; second, the user has to be supported in not getting lost within the Browser display; and third, nodes and links have to be automatically placed within the Browser window. We discuss each of these issues in turn below.

4.5.1. Reduction Strategies

In Section 3 above we discussed the notion of a *domain* within the instance repository. The Browser can display either an entire selected domain, or a sub-domain thereof created by choosing one of the following options:

(i) One type only. This displays only instances of the domain which are of a particular type. This strategy can be used if the user wishes to enter the domain through a specific type and wishes to see all current instances of that type in the domain. Often, particular types can be identified as 'natural' entry-points to the database;
(ii) Isolated nodes only. This displays only those instances of the domain which are not contained in the node-set of any other instance and which themselves do not contain any other instances. This strategy is particulary helpful for identifying 'stray' instances;
(iii) Non-nested nodes only. This displays only those instances which are not contained in any instance but which do themselves contain other instances. Again, such instances often turn out to be 'natural entry-points' to the database;
(iv) Nested nodes only. This displays instances which are both contained in some instance and themselves contain other instances;
(v) One instance only. This displays the instance that is currently being worked on within the Instance Editor, as well all the instances that it contains. This strategy is the default while a new instance is being created. Since the number of nodes that are being displayed is typically small the Browser can generate quick output and can assist the user by presenting an up-to-date picture of the data being edited.

4.5.2. Database Navigation

'Getting lost' in large graph data structures has been found to be a major problem ever since the advent of Hypertext databases [20]. Several techniques have been developed to assist the user when navigating through such databases, some of which our Browser makes use of. First, it maintains a backlog of the last n (default 20) instances the user has visited. This 'trail' is superimposed over the normal Browser output in a different colour so that the user can visually trace the path that was used to reach the current instance. All instances that are contained within the trail are always displayed in the Browser window, irrespective of whether they comply with the current reduction strategy or not. Second, the information stored in the backlog is accessible to the other components of the implementation architecture which can use it to support a 'go back' function (as the Instance Editor indeed does).

4.5.3. Node Placement

Node placement is one of the major problems that the Browser has to address. Generally, the Browser has to handle any number of links between any number of nodes. It has to create a clear picture of the domain structure, for example avoiding crossing of links as much as possible. Moreover, the display has to be produced within a acceptable amount of time and it has to be *repeatable.* Repeatability ensures that the user is presented with the same picture for the same domain. It is also desirable that minor changes in the domain, such as the insertion of a new isolated instance, result in only minor changes to the Browser representation.

A graph-drawing algorithm was recently described in [9] which deterministically produces satisfactory output. However, this algorithm is computationally intensive and in practice takes too long to compute its output for our purposes. In contrast,

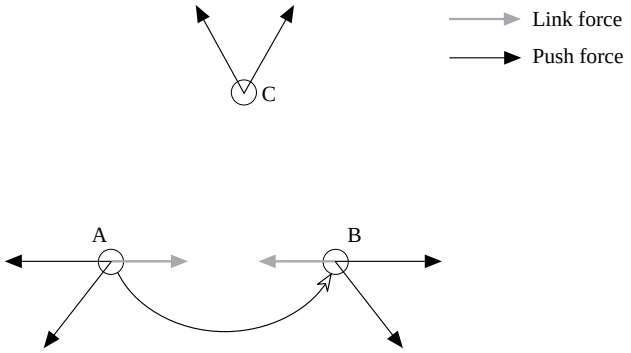

Figure 16. The nodes and the forces between them

our graph-drawing algorithm is *dynamic* in that it can be tailored to deliver a result after an arbitrarily short or long period of time, at the price that the result will be nearly optimal if the time provided is long and less optimal if the computation is to terminate within a shorter period of time. Our algorithm combines ideas from mechanics and self-organising networks [2, 13] and we now describe its salient features:

Firstly, two kinds of forces are defined between nodes:

(i) *push-forces,* $\bar{p}$, appear between any two nodes and force them apart, similar to the magnetic force between two charged particles whose charge is of same polarity;

(ii) *link-forces,* $\bar{l}$, are generated by a link between two nodes and force the nodes together, similar to the force exerted by a rubber-band.

Both kinds of forces are directed forces and can thus be computed using vector arithmetic. Figure 16 shows an example scenario with three nodes, two of which are linked, and all the forces acting on the nodes.

In our discussion below, we will denote the vector from a node a to a node b (i.e. $\bar{b} - \bar{a}$) by $\bar{\psi}_{a,b}$, the Euclidean distance between a and b by $|\bar{\psi}_{a,b}|$, and the unit vector from a to b by $\bar{\psi}^{\circ}_{a,b}$.

The size of each force is a function of the distance between the two nodes. The push-force on a node a that is generated by a node b can be defined as follows, where f is some function that is monotonic decreasing in its argument:

$$\bar{p}_{a,b} = -\bar{\psi}^{\circ}_{a,b} \cdot f(|\bar{\psi}_{a,b}|)$$

Note that the direction of the push-force is opposite to the direction of $\bar{\psi}^{\circ}_{a,b}$ i.e. the force is pushing the nodes apart.

The link-force on a node a that is generated by a link to a node b can similarly be defined as follows, where g is some function that is monotonic increasing in its argument:

$$\begin{aligned} \bar{l}_{a,b} &= \bar{\psi}^{\circ}_{a,b} \cdot g(|\bar{\psi}_{a,b}|) && \text{if } b \text{ is in the node-set of } a \\ &= 0 && \text{otherwise} \end{aligned}$$

Thus, the link-force acts in the same direction as $\bar{\psi}^{\circ}_{a,b}$.

It can be noted that for some value of $|\bar{\psi}_{a,b}|$, $f(|\bar{\psi}_{a,b}|) = g(|\bar{\psi}_{a,b}|)$. This is, effectively, the distance at which no net force acts on two nodes that are linked to each other. We call this distance the 'resting distance'.

Given a set of nodes, N, the vector $\bar{\Psi}_i$ gives the sum of all the forces acting on any node $a_i \in N$:

$$\bar{\Psi}_i = \sum_{j=1}^{|N|} (\bar{p}_{a_i,a_j} + \bar{l}_{a_i,a_j})$$

Finally, we define the *energy* of the system to be the sum of the scalars of all the forces acting onto all the nodes at any time:

$$\Lambda = \sum_{i=1}^{|N|} |\bar{\Psi}_i|$$

Λ is a measure of the 'orderedness' of the current state: a small value represents a state where unrelated nodes are far apart and where nodes that are linked are close to their resting distance. The problem of positioning a set of nodes can therefore be rephrased as finding a minimum for Λ. Our algorithm attempts to find a local minimum by iteration. At each step of the iteration, it allows each node a_i to be dragged away by the vector $\bar{\Psi}_i$ i.e. by the link- and push-forces currently acting upon it. Problems arise since the dynamic behaviour of the forces itself can only be approximated by taking 'snapshots' before each iteration, computing the new forces and then moving the nodes. Taking these snapshots at higher frequencies will improve the simulation-behaviour but slow down the entire process. To ensure that the algorithm will terminate, a *temperature*, t, is introduced into the system to act as a multiplicative factor to the size of the forces. The temperature is lowered at each iteration, until eventually it 'freezes' the movement of the nodes and the algorithm terminates. Hence, the definition of $\bar{\Psi}_i$ is modified to

$$\bar{\Psi}_{i,n} = t(n) \cdot \sum_{j=1}^{|N|} (\bar{p}_{a_i,a_j} + \bar{l}_{a_i,a_j})$$

where n denotes the current iteration and $t(n)$ is a suitably chosen temperature function, returning values in the range $0..1$. The following temperature function, which produces a temperature in the range $0..0.2$, has been found to give satisfactory results:

$$t(n) = \frac{0{\cdot}2}{1{\cdot}001^n}$$

In practice, the freezing point has been found to be around $0{\cdot}01$, which is reached after 5300 iterations. At $0{\cdot}01$, the value of Λ is almost always less than 1 pixel which means that no changes will be made to any node position. In fact, it has been found that the algorithm reaches this stage earlier in most cases, simply because the forces themselves have come to a minimum. Generally, by modifying the temperature function in such a way that the temperature decreases more quickly, the algorithm can be forced to reach a 'frozen' stage earlier. However, the resulting layout may not be as good.

Clearly, the initial placement of the nodes has a significant influence on the final result. This can easily be certified by simply considering a situation where all the nodes are initially placed on a straight line. In this case, the nodes always remain in a linear arrangement. Equally unfruitful is an initial placement where all the nodes are positioned at the same location since in this case no directions can be computed and

no movements can happen at all. Finally, since we want to obtain repeatability, a random initial placement is also ruled out. Thus, a *circular* initial placement is used, and nodes are initially placed at equal intervals along the circumference of a circle.

The outline of the entire placement algorithm is given below, where `push_force` is a suitable implementation of the function f, and `link_force` a suitable implementation of the function g (in practice $f(x) = 10000/x^2$ and $g(x) = x$ have been found to give satisfactory results):

```
PROCEDURE placement (N, L)          {N is the set of nodes and L the set of links}
VAR     p:ARRAY [1 . . |N|] OF VECTOR {the push force on each node}
        l:ARRAY [1 . . |N|] OF VECTOR {the link force on each node}

        initial_placement (N)       {generate the initial placement of the nodes}
        t=0·2                       {initialise the temperature}
        REPEAT
            t=t/1·001;              {decrement the temperature}
            FOR i=1 TO |N|          {reset the forces}
                p[i]=(0, 0)
                l[i]=(0, 0)
            FOR i=1 TO |N|          {compute the forces for each node}
                FOR j=i TO |N|      {consider each other node in turn}
                    p[i]=p[i]+t*push_force(i, j)
                    l[i]=l[i]+t*link_force(i, j, L)
                    p[j]=p[j]+t*push_force(j, i)
                    l[j]=l[j]+t*link_force(j, i, L)
            sum=0
            FOR i=1 TO |N|          {move the nodes}
                move(i, p[i]+l[i])
                sum=sum+SIZE(p[i])+SIZE(l[i])
        UNTIL sum<1                 {terminate if sum of movements is less
                                     than one pixel}
```

Figure 17 shows a typical positioning sequence. There are 10 iterations between each of the screen-dumps with the first representing the initial circular node placement. The entire process terminated after just 60 iterations in less than 2 seconds. Experience has shown that the algorithm rarely executes more than 150 iterations.

The maximum number of nodes that can comfortably be presented on one screen-full is around 25. If the number of instances within the selected domain and reduction strategy exceeds this number, the Browser performs a horizontal and vertical partitioning of the domain into multiple pages of output between which the user can move with simple menu-bar commands. The partitioning is determined by the links between the hypernodes: the isolated and non-nested hypernodes are partitioned into the first level of pages; for such page, its children pages contain the hypernodes nested within its hypernodes; and so on until all the hypernodes are allocated to a page.

5. Comparison with other Graph-Based Database Languages

Many other database languages have also used graphs as their underlying data structure. Several provide graphical query formulation facilities over the conceptual data model, for example [1, 8, 11]. Others, closer to our own work, aim to provide an

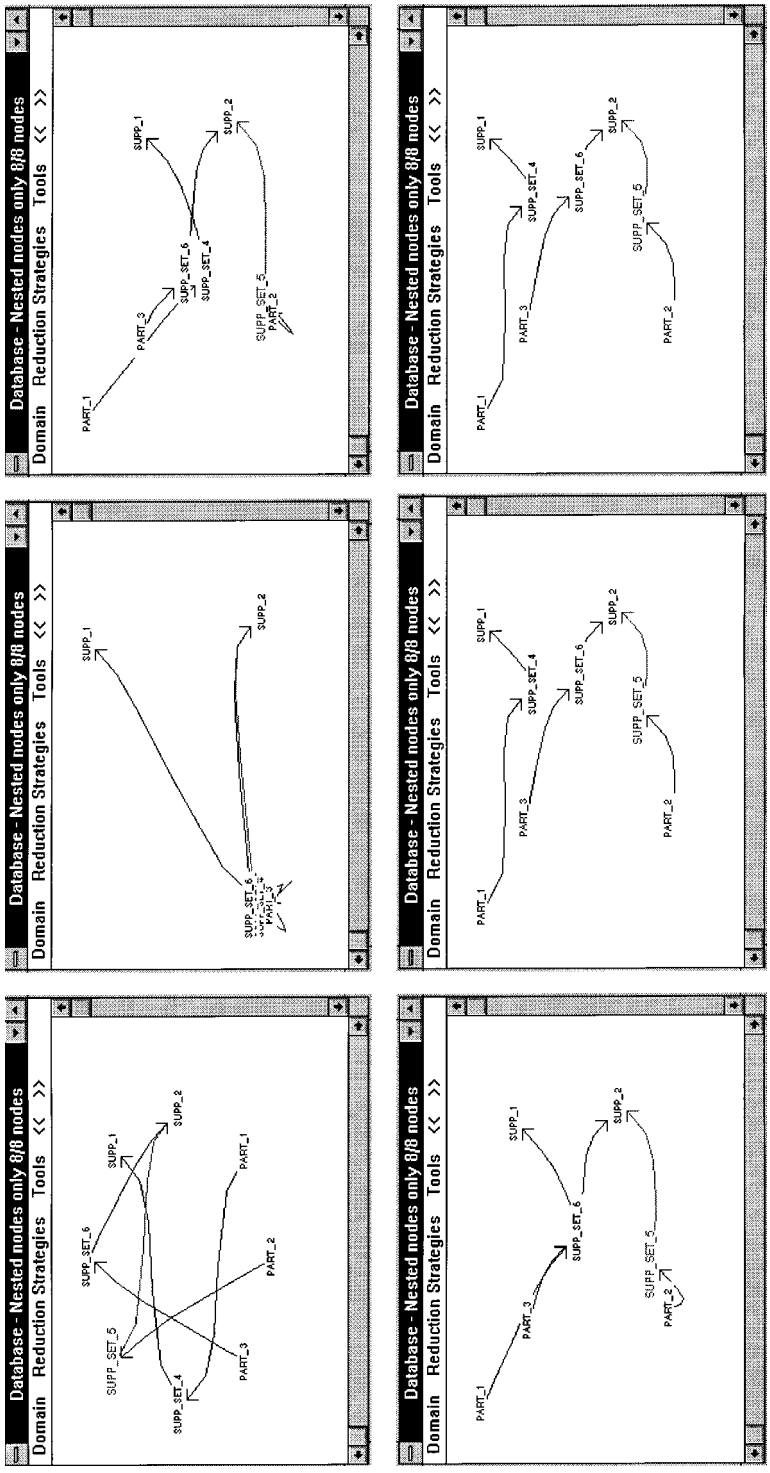

Figure 17. A typical positioning sequence

integrated approach to representing, querying and, possibly, updating both schema and data, for example [5, 6, 7, 10, 16, 17]. With the exception of [6], all these approaches regard the database as one flat graph as opposed to a set of nested graphs as in our case. Nesting is an extremely powerful abstraction and visualization technique which is inherent in the Hypernode Model and in Hyperlog. It allows two kinds of relationships to be represented graphically: links representing the containment of one hypernode within the node-set of another, and edges connecting two nodes in the node-set of the same hypernode. Nesting is also employed in Hy+ [6] whose graphs can have nodes, called *blobs,* which are sets of further nodes. However, hypernodes are more general since their nodes can be graphs. Other differences between other graph-based database languages and Hyperlog are as follows:

G+ [7] and its successor Graphlog [5] are query languages for the Hy+ environment [6]. In these languages queries are graphs whose edges are annotated with regular expressions c.f. Hyperlog which is rule based. These queries are matched against the database graph and return sub-graphs thereof. Database updates are not supported.

GOOD [10] is a graphically-represented functional data model with an associated transformation language. GOOD queries are also graphs, called *patterns,* which match sub-graphs of the database graph. Updates are specified by incorporating into patterns five graphically-represented primitive operations (for adding or deleting a node or an edge, and an operation called 'abstraction'). Programs consist of sequences of patterns. Hyperlog is a higher-level language and does not require the use of such primitives.

Hyperlog programs are strictly declarative i.e. no ordering is attached to rules. In contrast, G-Log [17] is a rule-based language whose rules are ordered. This allows the programmer procedural control of the execution semantics. In G-Log, different colouring is used to distinguish between positive and negative edges/nodes. Nesting is not supported: instead several types can be merged to form an instance which then appears as one entity. For the presentation of sizable amounts of data, G-Log must therefore utilize a textual presentation method.

Gql [16] provides a graphical representation of queries which are then translated into the functional database language FDL [18] for evaluation. Several queries can be combined into one query by linking them using negation, relational operators or aggregation. Computation is represented by annotating nodes with arithmetic expressions. Currently, Gql has no facilities for updates. In addition, the output to a Gql query is just the textural output of the underlying FDL.

To summarize, all the graphical query languages reviewed above rely to some extent on the use of predefined graphical symbols to obtain expressiveness and/or to solve ambiguities. Different colouring of edges or different shapes are typically used to uniquely define the intended semantic meaning. Being rule-based, Hyperlog can keep this overhead to a minimum. Thus, Hyperlog queries and programs consist of a set of graphs, and the only special symbols are negation of nodes and edges, and implication arrows (for programs only). Other novelties of our implementation are the automatic generation of edges within hypernodes, the algorithm for generating the Browser display, the storage and viewing of query output in the same way as the data, and the type-based creation of instances, queries and programs, thereby guaranteeing their type-correctness.

6. Conclusions

We have described the implementation of a graph-based query and update language called Hyperlog. We have in particular concentrated upon the graphical aspects of the implementation, including: the graphical creation of types (which are graphs) and the automatic drawing of the edges within them; the graphical creation of instances, queries and programs by instantiating currently defined types, thus ensuring their type-correctness; a novel algorithm for automatically displaying the database contents; and the storage and viewing of query output in the same way as the database.

A further novelty of our implementation is the close integration of the two main paradigms for interrogating databases, namely *declarative querying* and *navigation.* In particular, queries generate data which can be browsed and which can have further, 'what-if' style, queries posed against it.

As we mentioned in Section 2, the Hypernode Model can be used to visualize functional, relational and object-orientated data, while Hyperlog can be used to query and update this data at the same, very high, conceptual level. Thus, one of the important applications of Hyperlog is the visualisation and manipulation of existing heterogeneous databases.

References

1. M. Angelaccio, T. Catarci & G. Santucci (1990) QBD*: a graphical query language with recursion. *IEEE Transactions on Software Engineering* **16,** 1150–1163.
2. R. Beale & T. Jackson (1990) *Neural Computing—an Introduction.* Institute of Physics Publishing, London.
3. K. Benkerimi & A. Poulovassilis (1993) Semi-Naive Evaluation for Hyperlog, a graph-based language for complex objects. In: *1st International Workshop on Rules in Database Systems,* Edinburgh. Springer-Verlag Workshops in Computer Science, pp. 251–267.
4. S. Ceri, G. Gottlog & L. Tanca (1990) *Logic Programming and Databases.* Surveys in Computer Science, Springer-Verlag, Berlin.
5. M. P. Consens & A. O. Mendelzon (1990) Graphlog: a visual formalism for real life recursion. In: *ACM Symposium on Principles of Database Systems,* Nashville, Tennessee, pp. 404–416.
6. M. P. Consens & A. O. Mendelzon (1993) Hy+: A Hygraph-based query and visualization system. In: *ACM SIGMOD International Conference on Management of Data,* ACM Press, Denver, pp. 511–516.
7. I. F. Cruz, A. O. Mendelzon & P. T. Wood (1988) G+: Recursive Queries Without Recursion. In: *2nd International Conference on Expert Database Systems,* Tysons Corner, Virginia, ACM Press, Denver pp. 355–368.
8. B. Czedo, R. Elmasri & M. Rusinkiewicz (1990) A graphical data manipulation language for an Extended Entity-Relationship Model. *IEEE Computer* **23,** 26–36.
9. E. R. Gansner (1993) A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering* **19,** 214–230.
10. M. Gyssens M., J. Paredaens & D. V. Van Gucht (1990) A graph-oriented object model for database end-user interfaces. In: *ACM SIGMOD International Conference on the Management of Data,* Atlantic City, New Jersey, ACM Press, Denver pp. 24–33.
11. H. Kangassalo (1988) CONCEPT D: A graphical language for conceptual modelling and data base use. In: *IEEE 1988 International Workshop on Visual Languages,* Pittsburgh, ACM Press, Denver.
12. W. Kim (1990) Object-oriented databases: definition and research directions. *IEEE Transactions on Knowledge and Data Engineering* **2,** 327–341.
13. T. Kohonen (1990) Self Organisation and Associative Memory, Springer-Verlag.
14. M. Levene & A. Poulovassilis (1990) The Hypernode Model and its associated query

language. In: *5th Jerusalem Conference on Information Technology* (*JCIT-5*). IEEE Computer Society Press, Los Alamitos, CA, pp. 520–530.
15. M. Levene, A. Poulovassilis, K. Benkerimi, S. Schwartz & E. Tuv (1993) Implementation of a Graph-Based Data Model for Complex Objects. *ACM SIGMOD Record* **22**, 26–31.
16. A. Papantonakis & P. King (1994) Gql, a declarative graphical query language based on the functional data model. In: *Advanced Visual Interfaces Workshop,* Bari, Italy, pp. 133–122.
17. J. Paredaens, P. Peelman & L. Tanca (1991) G-Log: A Declarative Graphical Query Language. In: *2nd International Conference on Deductive and Object-Oriented Databases,* Munich. Springer-Verlag LNCS 566, Berlin, pp. 108–127.
18. A. Poulovassilis & P. King (1990) Extending the Functional Data Model to computational conpleteness. In: *International Conference on Extending Database Technology* (*EDBT-90*), Venice. LNCS 416, Springer-Verlag, Berlin, pp. 75–91.
19. A. Poulovassilis & M. Levene (1994) A nested-graph model for the representation and manipulation of complex objects. *ACM Transactions on Information Systems* **12**, 35–68.
20. I. Ritchie (1989) HYPERTEXT—moving towards large volumes. *The Computer Journal* **32**, 516–523.
21. D. F. Rogers & J. A. Adams (1976) *Mathematical Elements for Computer Graphics.* McGraw-Hill, London.
22. E. Tuv, A. Poulovassilis & Levene M. (1992) A storage manager for the hypernode model. In: *10th British National Conference on Databases,* Aberdeen. Springer-Verlag LNCS 618, Berlin, pp. 59–77.