A Graph-Based Data Model and its Ramifications

Mark Levene and George Loizou

Abstract- Currently database researchers are investigating new data models in order to remedy the deficiences of the flat relational model when applied to non-business applications. Herein we concentrate on a recent graph-based data model called the hypernode model. The single underlying data structure of this model is the hypernode which is a digraph with a unique defining label. We present in detail the three components of the model, namely its data structure, the hypernode, its query and update language, called HNQL, and its provision for enforcing integrity constraints. We first demonstrate that the said data model is a natural candidate for formalising hypertext. We then compare it with other graph-based data models and with set-based data models. We also investigate the expressive power of HNQL. Finally, using the hypernode model as a paradigm for graph-based data modelling, we show how to bridge the gap between graph-based and set-based data models, and at what computational cost this can be done.

Index Terms- graph-based data model, set-based data model, hypernode database, hypernode functional dependency, hypertext, query and update language, computable update, non-well-founded sets.

I. INTRODUCTION

Relational DataBase Management Systems (DBMSs) are currently dominating the commercial database market-place. The *flat relational model* (commonly known as the relational model) has been advocated by Codd since the early 1970's [CODD70]. It has taken about 20 years for the relational model to attain its present dominant position! Relational DBMSs have been developed with the traditional business data processing applications in mind, such as: banking, payroll and inventory control systems. The units of data needed for these applications are typically small and have a simple flat structure. Furthermore, the operations performed on these units of data are relatively straightforward and do not, in general, involve making recursive inferences.

In recent years there has been a growing demand to use databases in applications beyond the traditional business applications, such as: Computer Aided Software Engineering (CASE), hypertext, knowledge base systems, Computer Aided Design (CAD), image processing, scientific data (such as satellite data) analysis and geographical data analysis. In these applications the

M. Levene is with the Department of Computer Science, University College London, Gower Street, London WC1E 6BT, U.K., E-Mail address: M.Levene@uk.ac.ucl.cs.

G. Loizou is with the Department of Computer Science, Birkbeck College, University of London, Malet Street, London WC1E 7HX, U.K., E-Mail address: G.Loizou@uk.ac.bbk.cs.

units of data are typically larger and more complex than in the traditional business applications, i.e. they are complex objects whose structure may be hierarchical or may have a more general digraph structure. Furthermore, the operations needed to manipulate these complex objects may not be straightforward to define and may involve making recursive inferences. As an example, consider storing a VLSI chip layout in a database and defining the operations for modifying and testing the chip. As another example, consider a library which would like to have available on-line papers from scientific journals in a particular subject area such as Computer Science. The task of organising the text included in individual papers in a manner that allows readers to browse and query the text in a very flexible manner, and the task of creating the appropriate references between papers cannot be carried out easily by using the relational model.

Currently database researchers are actively investigating new data models in order to be able to manage efficiently applications not easily modelled using the relational approach, and are implementing prototype DBMSs based on these new models. There are two main categories of new data models that are being developed:

(1) *set-based* data models, such as the *nested relational model* [LEVE92, PARE89, THOM86, VANG88], which extend the traditional relational model, and

(2) *graph-based* data models, such as the *hypernode model* [LEVE90] presented herein, which build upon the traditional hierarchical and network data models [ULLM88].

Hereinafter we concentrate on a graph-based data model, namely the hypernode model. Overall, less research has been carried out on graph-based data models and as yet there is no agreement within the database community on a single graph-based data model. In contrast, the relational model and its nested relational counterpart have been extensively investigated and provide adequate formalisms for the development of set-based DBMSs.

We now informally introduce the three components of the hypernode model. The single underlying data structure of the hypernode model is the hypernode, which is an equation of the form G = (N, E) such that (N, E) is its digraph and G is its unique defining label. A hypernode database is a finite set of hypernodes.

In Fig. 1 we illustrate part of a simple hypernode database, which models a simple airline reservation system. The hypernode, whose defining label is AIRLINES, contains the defining labels of other hypernodes which describe the various airlines, and the hypernode, whose defining label is PASSENGERS, contains the defining labels of other hypernodes which describe the booking information pertaining to passengers. The hypernode with defining label FLIES represents a relationship telling us with which airline a particular passenger is flying. We note that labels are denoted by strings beginning with an uppercase letter.

In Fig. 2 we show the details of some of the passenger hypernodes. We note that atomic values are denoted by strings surrounded by double quotes and attribute names by strings

beginning with "$". We further observe that the attribute name $dependent in a passenger hypernode is used to reference other passengers who in some unspecified way depend on this passenger; for example, PASS2 and PASS3 may depend on PASS1 to drive them to the airport. These references between hypernodes can be viewed conceptually as a part-of relationship or alternatively as encapsulating the data represented in the referenced hypernodes. We use the distinguished atomic value *null* to indicate that a "value exists but is unknown". We note that we can also model incomplete information of the type "value does not exist" by isolated attribute names. For example, if we delete the arc ($dependent, *null*) and the node *null* from the digraph of the hypernode, whose defining label is PASS3, our interpretation changes from "PASS3 has a dependent which is unknown" to "there does not exist a dependent of PASS3".

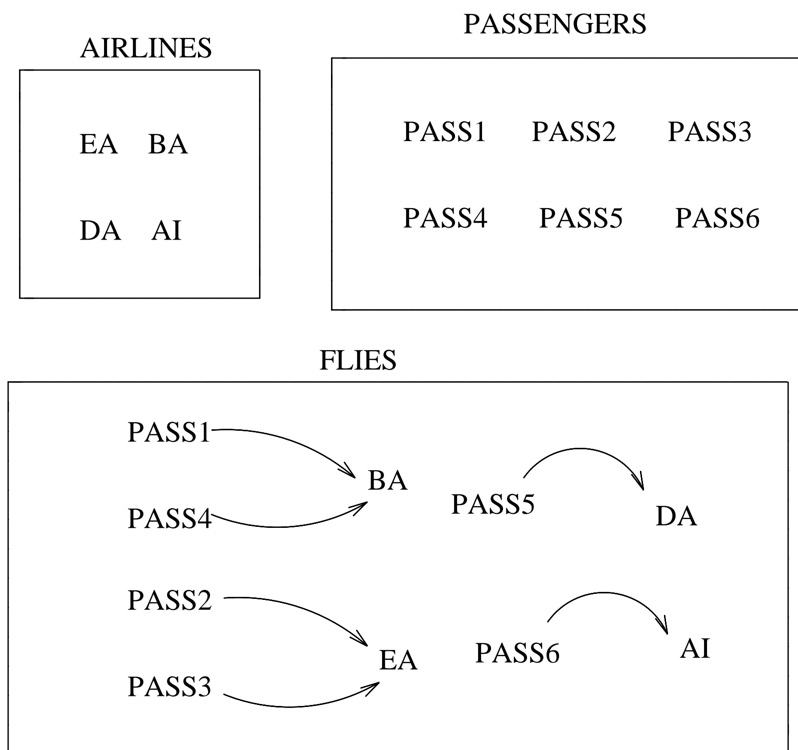

Fig. 1. Part of a passengers and airlines hypernode database.

The query and update language (or alternatively, the database language) for the hypernode model, presented herein, is called *HyperNode Query Language* (HNQL). HNQL consists of a basic set of operators for declarative querying and updating of hypernodes. In addition to the standard deterministic operators we provide several non-deterministic operators (cf. [ABIT90]), which arbitrarily choose a member from a set. Examples of the need for such non-determinism are: choosing an arbitrary seat number for a passenger on a given flight or choosing an arbitrary referenced paper on a specific topic from a given set of references. HNQL is further extended in a procedural style by adding to the said set of operators an assignment construct, a sequential

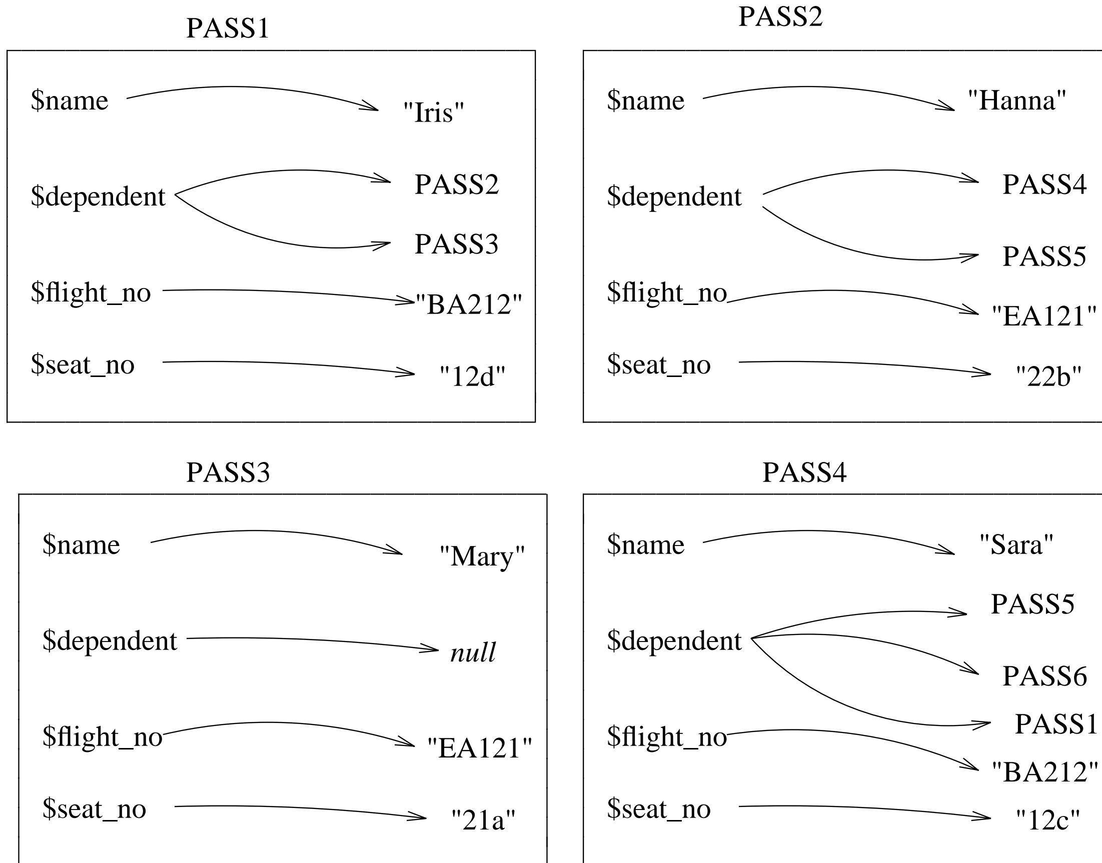

Fig. 2. Some of the passengers in the hypernode database.

composition construct, a conditional construct for making inferences and, finally, for loop and while loop constructs for providing iteration (or equivalently recursion) facilities.

Finally, we equip the hypernode model with an integrity constraint called the *Hypernode Functional Dependency* (HFD). HFDs incorporate into the hypernode model Functional Dependencies (FDs) from the relational model [PARE89, ULLM88] by using a graph-theoretic formalism.

We now very briefly overview some of the ramifications of the hypernode model which are given in detail in the paper. We demonstrate that it is a natural candidate for formalising hypertext [CONK87, NIEL90] due to its support of a general-purpose digraph constructor and its ability to support both navigation and declarative querying via HNQL. We then go on to investigate the expressive power of HNQL in terms of the class of transformations from databases to databases (termed *computable updates*) that can be expressed in HNQL. We present two classes of computable updates and discuss the expressive power of HNQL with respect to these two classes. Finally, using the hypernode model as a paradigm for graph-based data modelling, we show that it is possible to bridge the gap between graph-based and set-based data models. This can be

achieved by a transformation from hypernodes to *non-well-founded* sets [ACZE88, BARW91]. Unfortunately, such a transformation is shown to be at least as hard as testing for isomorphism of digraphs [BUCK90], whose complexity is, in general, an open problem [GARE79].

The rest of the paper is organised as follows. In Section II we present the three components of the hypernode model. In Section III we show that the hypernode model can provide an underlying formalism for hypertext. In Section IV we compare the hypernode model with other graph-based data models and with set-based data models. In Section V we discuss the expressive power of HNQL. In Section VI we show how to bridge the gap between graph-based and set-based data models. Finally, in Section VII we give our concluding remarks and indicate further research problems to be solved.

II. THE HYPERNODE MODEL

In this section we present the *hypernode model* which builds on the traditional graph-based models, i.e. the hierarchical and network data models [ULLM88]. In particular, in Section II-A we present the single underlying data structure of the model, namely the hypernode. In Section II-B we present the query and update language of the model, i.e. HNQL, and in Section II-C we show how FDs can be incorporated into the model in the form of HFDs.

A. Hypernodes and Hypernode Databases

The underlying data structure of the hypernode model is the *hypernode*, which is used to represent real-world objects. We begin by recalling the definition of a directed graph (or simply a *digraph*) [BUCK90]. A digraph is an ordered pair (N, E), where N is a finite set of nodes and E $\subseteq$ (N $\times$ N) is a set of ordered pairs of nodes from N, which are called arcs (also known as directed edges).

We use the following terminology for a digraph (N, E). An arc (n, m) $\in$ E is said to be *incident* with each of its two nodes n and m. We call n the *anchor* and m the *destination*. We also say that n is *adjacent to* m and that m is *adjacent from* n. The *indegree* of a node n $\in$ N is the number of nodes adjacent to n and the *outdegree* of n is the number of nodes adjacent from n. A node with no incident arcs is said to be *isolated*.

We assume the following two disjoint countable domains of constants are available. Firstly we have a domain of *Labels* **L** whose elements are denoted by strings beginning with an uppercase letter (excluding X and Y since these are used to denote variables). Secondly we have a domain of *Primitive nodes* **P** which is partitioned into two disjoint domains one of *Atomic Values*, **AV**, and the other of *Attribute Names* (or simply attributes), **AN**. We denote atomic values by strings surrounded by double quotes and attributes by strings beginning with "$". We also assume that the domain of atomic values **AV** contains a distinguished value *null* meaning "value exists but is unknown".

A hypernode is now defined to be an equation of the form:

$$G = (N, E)$$

where $G \in \mathbf{L}$ is termed the *defining label* of the hypernode (or simply the label of the hypernode when no ambiguity arises) and (N, E) is a digraph, termed *the digraph of the hypernode* (or simply the digraph of G), such that $N \subseteq (\mathbf{P} \cup \mathbf{L})$.

We impose the following syntactic restrictions on the arcs of a hypernode, G = (N, E):

(E1) the indegree of nodes $n \in (\mathbf{AN} \cap N)$, i.e. of nodes that are attributes, is zero.

(E2) the outdegree of nodes $n \in (\mathbf{AV} \cap N)$, i.e. of nodes that are atomic values, is zero.

(E3) if $(n, m) \in E$ and $n \in (\mathbf{L} \cap N)$, i.e. the anchor node of the arc is a label, then $m \in (\mathbf{L} \cap N)$, i.e. the destination node of the arc is also a label.

In order to explain the motivation behind the above restrictions we take the approach of the Entity-Relationship model [CHEN76] which asserts that the real world can be described by *entities* (or objects which in our case are hypernodes), which are in turn represented by a set of attributes and their values, and by *relationships* between entities.

We observe that an arc set of a digraph can be viewed as a (binary) relation on the nodes which are incident on its arcs. Thus, the semantics of restriction E1 are that attributes cannot be in the range of the relation induced by the arc set. Furthermore, an arc whose anchor is an attribute represents an attribute-value pair (i.e. a property) whose destination node is its value, the value being either an atomic value or a label. Thus, when an arc is incident with an attribute this attribute must be the anchor of the arc. The semantics of restriction E2 are that atomic values cannot be in the domain of the relation defined by the arc set. Thus, when an arc is incident with an atomic value this value must be the destination of the arc. Finally, the semantics of restriction E3 are that when a label is in the domain of the relation defined by the arc set then an arc incident with this label represents a relationship between two hypernodes, i.e between two objects. Thus, a relationship between two hypernodes can be represented by an arc which is incident with their defining labels. We observe that conceptually this kind of relationship can be viewed as a *referential* relationship (see relational links [DERO89]).

It can easily be verified that the hypernodes shown in Fig. 1 and Fig. 2 satisfy restrictions E1, E2 and E3. As was discussed in the introduction these hypernodes model part of a simple airline reservation system detailing information about passengers and indicating with which airline a particular passenger is flying. We note that each arc in the hypernode with the defining label FLIES in Fig. 1 represents a referential relationship and that each arc in the passenger hypernodes of Fig. 2 represents an attribute-value pair.

A *hypernode database* (or simply a database), say HD, is a finite set of hypernodes satisfying the following three conditions:

(H1) no two (distinct) hypernodes in HD have the same defining label.

(H2) for any label, say G, in the node set of a digraph of a hypernode in HD there exists a hypernode in HD whose defining label is G.

(H3) every attribute name has the same meaning in all the node sets (of all the digraphs) of all the hypernodes in HD in which this attribute name appears.

Given a database, HD, we denote by LABELS(HD) the set of labels appearing in the hypernodes of HD, by PRIM(HD) the set of primitive nodes appearing in the hypernodes of HD, and by ATT(HD) the set of attributes appearing in HD, i.e. PRIM(HD) $\cap$ **AN**.

We note that condition H1 above corresponds to the *entity integrity* requirement of [CODD79], since each hypernode can viewed as representing a real-world entity. In object-oriented terminology [KIM90] labels are unique and serve as system-wide object identifiers, assuming that all of the hypernodes known to the system are stored in a single database. Similarly, condition H2 corresponds to the *referential integrity* requirement of [CODD79], since it requires that only existing entities be referenced. This implies that a relationship between two hypernodes can also be represented in terms of a reference from one hypernode to the other (rather than a reference via an arc between two labels in the digraph of a hypernode). We observe that conceptually this kind of relationship can be viewed as a *part-of* relationship, which provides the hypernode model with inherent support for data encapsulation (see inclusion links [DERO89]). Condition H3 corresponds to the *Universal Relation Schema Assumption* originating from the *Universal Relation* model [LEVE92, MAIE84]. For example, if the attribute, $title, means the title of a document it cannot also mean the title of an author of a document. We note that condition H3 can always be enforced by the renaming of attributes when a conflict does arise. For example, we could have the attribute, $dtitle, meaning the title of a document, and the attribute, $atitle, meaning the title of an author of a document.

It can easily be verified that the hypernodes shown in Fig. 1 and Fig. 2 comprise a portion of a hypernode database (if we add hypernodes for PASS5, PASS6, EA, BA, DA and AI, whose node sets do not include any new labels, we would then have a database satisfying conditions H1, H2 and H3). We note that by condition H1 each hypernode representing one of the objects in the database has a unique label. Furthermore, the defining labels of the passenger hypernodes are part-of the hypernode with the defining label PASSENGERS. Thus, by condition H2 there must be one hypernode in the database for each passenger. Finally, we note that by condition H3 each attribute included in the passenger hypernodes shown in Fig. 2 plays a unique role.

The *Hypernode Accessibility Graph* (HAG) of a hypernode G = (N, E) $\in$ HD (or simply the HAG of G, whenever HD is understood from context) is the digraph telling us which hypernodes in HD are part-of (or encapsulated in) the hypernode with the defining label G, when considering

part-of as a transitive relationship (cf. composite objects [KIM90]).

Formally, we define the HAG of G, denoted by (N_G, E_G), as the minimal digraph which is constructed from hypernodes in HD as follows:

(1) $G \in N_G$, and G is a distinguished node called the *root* of (N_G, E_G);

(2) if $G' \in N_G$ and $G' = (N', E') \in$ HD (such a hypernode must exist by condition H2), then $(\mathbf{L} \cap N') \subseteq N_G$ and $\forall\, n' \in (\mathbf{L} \cap N'), (G', n') \in E_G$.

We note that, in general, the HAG of G may be cyclic. In Fig. 3 we illustrate the HAG of PASS1, where the hypernode with defining label PASS1 is shown in Fig. 2. We note that the HAG of PASS1 is cyclic and thus PASS4 is part-of PASS1 and PASS1 is part-of PASS4, indicating that PASS1 and PASS4 depend on each other.

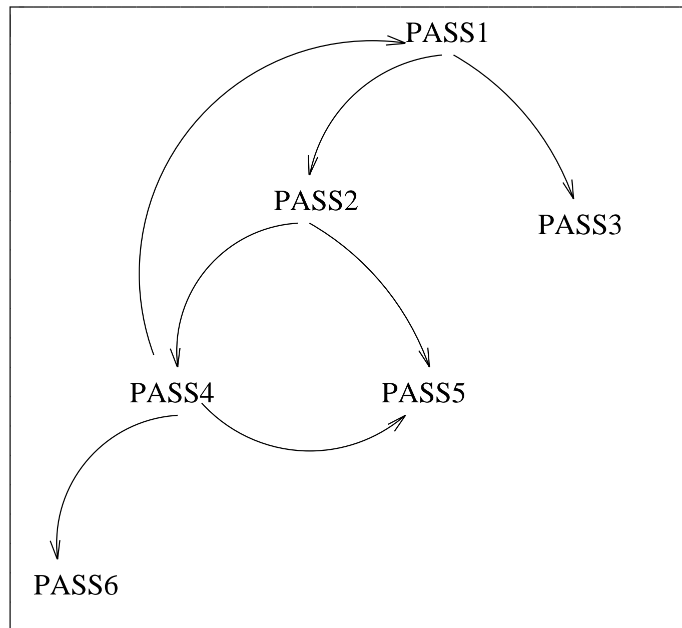

Fig. 3. The HAG of PASS1.

In order to simplify the presentation we assume that hypernodes are untyped, i.e. we do not put any further constraints on the structure of hypernodes. Thus, hypernodes are dynamic in the sense that nodes and arcs in hypernodes can be updated subject only to all of the above restrictions. In this approach we do not classify entities according to the entity set to which they belong but rather consider entities to be *classless* [ULLM91] (cf. [RICH91]), i.e. belonging to a single set of entities. In particular, all the available hypernodes are members of a single database. (Types give us a means of defining database schemas and of enforcing further constraints on the structure and content of hypernodes. An extension of the hypernode model to deal with typed hypernodes and typed databases can be found in [POUL92].)

B. A Query and Update Language for Hypernodes

We now introduce a query and update language for the hypernode model, called *HyperNode Query Language* (HNQL). HNQL consists of a basic set of operators for declarative querying and updating of hypernodes. HNQL is further extended in a procedural style by adding to the basic set of operators an assignment construct, a sequential composition construct, a conditional construct for making inferences and, finally, for loop and while loop constructs for providing iteration (or equivalently recursion) facilities.

Apropos of HNQL we assume that a countable domain of variables, **V**, is available, and such that $\mathbf{V} \cap \mathbf{L} = \mathbf{V} \cap \mathbf{P} = \emptyset$. We denote variables by strings beginning with the uppercase letters X or Y. Variables in HNQL are untyped, i.e. their values range over the union of the domains of primitive nodes and labels.

From now on we assume that HD is a hypernode database and that all the operators we define are to be evaluated with respect to HD. We also assume that the label NULL $\notin$ LABELS(HD) is reserved in order to return an error code when necessary. Notationally, we will use strings beginning with the lowercase letter, v, to denote either a label or a primitive node and strings beginning with the uppercase letter G to denote labels only.

The following four operators update hypernodes in the database, HD:

(1) insert_node(G, v) returns G if G = (N, E) $\in$ HD, and as a side effect v is inserted into N, i.e. $N := N \cup \{v\}$; otherwise NULL is returned.

(2) delete_node(G, v) returns G if G = (N, E) $\in$ HD and $\forall\, v' \in N$ there is no arc $(v, v') \in E$ or $(v', v) \in E$, and as a side effect v is deleted from N, i.e. $N := N - \{v\}$; otherwise NULL is returned.

(3) insert_arc(G, v_1, v_2) returns G if G = (N, E) $\in$ HD and $v_1, v_2 \in N$, and as a side effect (v_1, v_2) is inserted into E, i.e. $E := E \cup \{(v_1, v_2)\}$; otherwise NULL is returned.

(4) delete_arc(G, v_1, v_2) returns G if G = (N, E) $\in$ HD and $(v_1, v_2) \in E$, and as a side effect (v_1, v_2) is deleted from E, i.e. $E := E - \{(v_1, v_2)\}$; otherwise NULL is returned.

The following two operators add or remove hypernodes from the database, HD:

(1) create() returns an arbitrary new label G such that G $\notin$ (LABELS(HD) $\cup$ {NULL}), and as a side effect $G = (\emptyset, \emptyset)$ is added to HD, i.e. HD := HD $\cup$ $\{G = (\emptyset, \emptyset)\}$.

(2) destroy(G) returns the label G if $G = (\emptyset, \emptyset) \in$ HD and for no hypernode $G' = (N', E') \in$ HD is it true that $G \in N'$, and as a side effect $G = (\emptyset, \emptyset)$ is removed from HD, i.e. HD := HD $-$ $\{G = (\emptyset, \emptyset)\}$; otherwise NULL is returned.

The following five predicates provide membership tests for a node or an arc being contained

in a given hypernode, for a defining label of a hypernode being in the database, HD, or for the digraph of a hypernode to contain a given node or a given arc:

(1) $v \in$ nodes(G) returns true if $G = (N, E) \in$ HD and $v \in N$; otherwise false is returned.

(2) $(v_1, v_2) \in$ arcs(G) returns true if $G = (N, E) \in$ HD and $(v_1, v_2) \in E$; otherwise false is returned.

(3) $G \in$ db() returns true if $G = (N, E) \in$ HD; otherwise false is returned.

(4) $G \in$ node_to_graphs(v) returns true if $G = (N, E) \in$ HD and $v \in N$; otherwise false is returned.

(5) $G \in$ arc_to_graphs(v_1, v_2) returns true if $G = (N, E) \in$ HD and $(v_1, v_2) \in E$; otherwise false is returned.

We also allow the two equality tests: $v_1 = v_2$ for nodes, and $(v_1, v_2) = (v_3, v_4)$ for arcs, which return true or false as the case may be.

We define a *simple condition* to be either a membership test or an equality test. A *condition* is now defined to be either a simple condition, the parenthesising of a condition used for grouping purposes, the negation of a condition, say *cond*, denoted by !*cond*, or the conjunction of two conditions, say $cond_1$ and $cond_2$, denoted by $cond_1$ & $cond_2$.

The following five non-deterministic operators can be used to arbitrarily choose a node or an arc contained in a given hypernode, or arbitrarily choose a defining label of a hypernode in the database, HD, or one containing a given node or a given arc:

(1) any_node(G) returns an arbitrary node $v \in N$ if $G = (N, E) \in$ HD and $N \neq \emptyset$; otherwise NULL is returned.

(2) any_arc(G) returns an arbitrary arc $(v_1, v_2) \in E$ if $G = (N, E) \in$ HD and $E \neq \emptyset$; otherwise (NULL, NULL) is returned.

(3) any_label() returns an arbitrary label G such that $G = (N, E) \in$ HD, if HD $\neq \emptyset$; otherwise NULL is returned.

(4) node_to_any_graph(v) returns an arbitrary label G such that $G = (N, E) \in$ HD and $v \in N$; otherwise NULL is returned.

(5) arc_to_any_graph(v_1, v_2) returns an arbitrary label G such that $G = (N, E) \in$ HD and $(v_1, v_2) \in E$; otherwise NULL is returned.

Hereafter we assume that all variables in HNQL have a current value, which is either a label or a primitive node; these are always initialised to have the value NULL. Thus, we extend our earlier notation to allow strings beginning with the letters v or G to denote the current value of a variable when appropriate. We now define an *assignment* statement to be an expression of the

form:

lvalue := rvalue

where lvalue is a variable or a pair of variables, and rvalue is a constant, or a variable, or any of the possible pairs of these two, or one of the HNQL operators defined so far.

The semantics of an assignment statement are that the current value of lvalue becomes the result of evaluating rvalue on the current state of the hypernode database, HD (and possibly updating HD as a side effect). We note that evaluating a constant on HD returns the constant itself and that evaluating a variable on HD returns its current value. We assume that if the assignment is undefined, for example, when trying to assign a pair of constants to a variable, or a constant to a pair of variables, then lvalue is assigned the value NULL or (NULL, NULL), respectively. This is consistent with the standard destructive assignment of imperative programming languages such as Pascal [JENS85] for defining the value of a variable.

Statements (which can be assignment statements or one of the other kinds of statements defined subsequently) can be composed sequentially using ";" as a statement separator. Furthermore, we use the keywords **TB** (transaction begin) and **TE** (transaction end) to delimit such compound statements in analogy to the *begin* and *end* keywords used in Pascal. For convenience we may omit **TB** and **TE** whenever a compound statement contains only a single statement. We note that since a compound statement is a statement, nesting of compound statements is made possible.

The compound statment, shown in Fig. 4, deletes the arc ($dependent, *null*) and the node *null* from the hypernode with defining label PASS3 and then inserts the node PASS5 and the arc ($dependent, PASS5) into this hypernode. Finally, an arbitrary arc is deleted from the hypernode with defining label FLIES.

The syntax of a conditional statement is defined as follows:

if condition **then**
TB compound statement **TE**

The semantics of a conditional statement are that if the condition evaluates to true on the current state of the database, HD, then the compound statement is executed on the current state of HD. On the other hand, if the condition evaluates to false then the compound statement is not executed at all.

The conditional statement, shown in Fig. 5, deletes the arc ($dependent, *null*) and the node *null* from the hypernode with defining label PASS3 and then inserts the node PASS2 and the arc ($dependent, PASS2) into it, if both PASS2 and PASS3 are flying on flight_no "EA121" and PASS2 is not already a dependent on PASS3.

```
TB
  X := delete_arc(PASS3, $dependent, null);
  X := delete_node(PASS3, null);
  X := insert_node(PASS3, PASS5);
  X := insert_arc(PASS3, $dependent, PASS5);
  (Y1, Y2) := any_arc(FLIES);
  X := delete_arc(FLIES, Y1, Y2);
TE
```

Fig. 4. An example of a compound statement.

```
if PASS2 ∈ arc_to_graphs($flight_no, "EA121") &
      PASS3 ∈ arc_to_graphs($flight_no, "EA121") &
      !PASS3 ∈ arc_to_graphs($dependent, PASS2) then
  TB
      X1 := delete_arc(PASS3, $dependent, null);
      X1 := delete_node(PASS3, null);
      X1 := insert_node(PASS3, PASS2);
      X1 := insert_arc(PASS3, $dependent, PASS2);
  TE
```

Fig. 5. An example of an if statement.

We next define two types of loop: for loops, which give us a *bounded looping* construct, and while loops, which give us an *unbounded looping* construct [CHAN88].

The syntax of a for loop is defined as follows:

for_all for_predicate **do**
TB compound statement **TE**

where for_predicate is one of the following five membership testing predicates: X ∈ nodes(G), (X1, X2) ∈ arcs(G), X ∈ db(), X ∈ node_to_graphs(v) and X ∈ arc_to_graphs(v_1, v_2).

The semantics of a for loop are now described. Firstly, the for_predicate is evaluated on the current state of the database, HD, prior to the execution of the for loop. The evaluation is effected once for each possible substitution of the variables in the for_predicate with values from LABELS(HD) ∪ PRIM(HD). Thereafter the compound statement is executed synchronously in parallel on the current state of HD once for each time the for_predicate evaluates to true with the

evaluation as indicated above. (We note that the semantics of a compound statement being executed in parallel are that the statements, which comprise this compound statement, are also to be executed synchronously in parallel.) We further observe that the compound statement is always executed only a finite number of times, i.e. the looping is bounded.

The for loop, shown in Fig. 6, modifies flight number "BA212" to "BA345" for all passengers in the database.

The syntax of a while loop is defined as follows:

while changes do
TB compound statement **TE**

```
for_all X1 ∈ nodes(PASSENGERS) do
  for_all (Y1, Y2) ∈ arcs(X1) do
  TB
    if (Y1, Y2) = ($flight_no, "BA212") then
    TB
      X2 := delete_arc(X1, Y1, Y2);
      X2 := delete_node(X1, Y2);
      X2 := insert_node(X1, "BA345");
      X2 := insert_arc(X1, Y1, "BA345");
    TE
  TE
```

Fig. 6. An example of a for loop.

The semantics of a while loop are that the compound statement is repeatedly executed on the current state of the database HD until no further changes are effected on the current state of HD. That is, the compound statement is executed until a fixpoint is attained. We observe that, in general, a while loop may not terminate (since a fixpoint may not be attainable), i.e. the number of times the compound statement is executed may be unbounded.

The while loop, shown in Fig. 7, transitively closes the digraph of a hypernode, G = (N, E). Note that we have omitted **TB** and **TE**, since this while loop comprises a single statement.

An HNQL *program* is now defined to be a compound statement terminated by a full-stop, i.e. it is a sequential composition of one or more of the above kinds of statements (including a compound statement itself). The HNQL program, shown in Fig. 8, oscillates between updating the digraph of a hypernode G = (N, E) to be irreflexive and updating it to be reflexive. We note that this program does not terminate.

```
while changes do
  for_all (X1, X2) ∈ arcs(G) do
    for_all (X3, X4) ∈ arcs(G) do
      if X2 = X3 then
        X5 := insert_arc(G, X1, X4);
```

Fig. 7. An example of a while loop.

```
TB
  while changes do
  TB
    for_all X1 ∈ nodes(G) do
        if (X1, X1) ∈ arcs(G) then
          X2 := delete_arc(G, X1, X1);
    for_all X1 ∈ nodes(G) do
        if !(X1, X1) ∈ arcs(G) then
          X2 := insert_arc(G, X1, X1);
  TE
TE.
```

Fig. 8. An example of an HNQL program.

C. Hypernode Functional Dependencies

Functional Dependencies (FDs) are by far the most common integrity constraint in the real world [PARE89, ULLM88] and the notion of key (derived from a given set of FDs) [CODD79] is fundamental to the relational model. FDs have also been extended to nested relations in [LEVE92, THOM86, VANG88]. Essentially, by allowing attribute domains to be relation-valued (i.e. nested relations) FDs are capable of modelling both single-valued and multi-valued data dependencies. In [LEVE91] it was shown that such extended FDs can be naturally incorporated into a hypergraph-based data model which was the precursor of the hypernode model. FDs have also been incorporated into the graph-based model GOOD [GYSS90] in the form of functional and multi-valued arcs. Finally, a more general type of FD, called a *path FD* (PFD), was defined in [WEDD92] for an object-oriented data model, wherein both the class schemas and the instances thereof are interpreted as digraphs. A sound and complete axiomatisation of PFDs was exhibited and it was also shown that, in general, if the schema is cyclic then there may be an infinite number of derived PFDs.

We now show how the concept of FDs can be incorporated into the hypernode model by using a graph-theoretic formalism.

We recall that a *subgraph*, (N', E'), of (N, E), is a digraph such that $N' \subseteq N$ and $E' \subseteq E$. The *induced subgraph* of (N, E) with node set S, denoted by induced(S, (N, E)), is the maximal subgraph of (N, E) whose node set is S [BUCK90].

Let HD be a hypernode database, $G = (N, E) \in HD$ and $A \subseteq (N \cap ATT(HD))$. We denote by adj(A) the set of attributes A together with the set of all nodes $m \in N$ that are adjacent from any node $n \in A$.

We next give a definition of a FD in HD. Informally a set of attributes, $A \subseteq ATT(HD)$, functionally determines another set of attributes, $B \subseteq ATT(HD)$, if for each attribute, $\$b \in B$, whenever the induced subgraphs, each with node set adj(A), of two digraphs of hypernodes in HD are equal, then the corresponding induced subgraphs, each with node set adj({$b}), of these two digraphs are also equal.

More formally, let $A, B \subseteq ATT(HD)$ be two sets of attributes. Then, the *Hypernode Functional Dependency* (HFD), $A \rightarrow B$, is satisfied in HD if $\forall$ $\$b \in B$ and for every pair of (not necessarily distinct) hypernodes $G_1 = (N_1, E_1)$, $G_2 = (N_2, E_2) \in HD$ such that

(1) $A, \{\$b\} \subseteq N_i$, for $i = 1,2$, and

(2) $induced(adj(A), (N_1, E_1)) = induced(adj(A), (N_2, E_2))$,

it is also the case that

(3) $induced(adj(\{\$b\}), (N_1, E_1)) = induced(adj(\{\$b\}), (N_2, E_2))$.

An example of an HFD holding in our simple airline reservation database is:

{$flight_no, $seat_no} → {$name, $dependent}

This HFD asserts that a passenger's seat number and flight number uniquely determine their name and their dependents. It can easily be verified that the passenger hypernodes in Fig. 2 satisfy this HFD.

In the following we assume that ATT(HD) is a fixed set of attributes, U, for any hypernode database, HD, that F is a set of HFDs and that $A \rightarrow B$ is a single HFD.

We denote the fact that a database HD satisfies F (respectively, by $A \rightarrow B$) by $HD \models F$ (respectively, $HD \models A \rightarrow B$). We say that F logically implies $A \rightarrow B$ (with respect to a class of hypernode databases), denoted by $F \models A \rightarrow B$, if and only if for every hypernode database HD in the given class if $HD \models F$ then $HD \models A \rightarrow B$.

An axiom system for HFDs (for a given class of hypernode databases) is a set of inference rules that can be used to derive HFDs from a given set F of HFDs. We denote by $F \vdash A \rightarrow B$ the

fact that either $A \rightarrow B \in F$ or $A \rightarrow B$ can be inferred from F by using one or more of the inference rules in a given axiom system for HFDs. We define the closure of a set of attributes, A, with respect to F, denoted by A^+, to be the set of all attributes such that $\$b \in A^+$ if and only if $F \vdash A \rightarrow \{\$b\}$. Finally, an axiom system is sound if $F \vdash A \rightarrow B$ implies that $F \models A \rightarrow B$ and it is complete if $F \models A \rightarrow B$ implies that $F \vdash A \rightarrow B$.

We now define an axiom system for HFDs with respect to the class of all hypernode databases.

(R1) Reflexivity: if $B \subseteq A \subseteq U$, then $F \vdash A \rightarrow B$.

(R2) Augmentation: if $F \vdash A \rightarrow B$ and $C \subseteq U$, then $F \vdash A \cup C \rightarrow B \cup C$.

(R3) Union: if $F \vdash A \rightarrow B$ and $F \vdash A \rightarrow C$, then $F \vdash A \rightarrow B \cup C$.

(R4) Decomposition: if $F \vdash A \rightarrow B$, then $\forall \$b \in B$, $F \vdash A \rightarrow \{\$b\}$.

We observe that the transitivity rule (i.e. if $F \vdash A \rightarrow B$ and $F \vdash B \rightarrow C$, then $F \vdash A \rightarrow C$), which is sound for FDs with respect to relational databases, is not sound as an inference rule for HFDs. Consider the following counterexample. Let HD be the database shown in Fig. 9, where A, C and D denote pairwise disjoint sets of attributes. It can easily be verified that $HD \models A \rightarrow D$, $HD \models D \rightarrow C$ but $HD \not\models A \rightarrow C$.

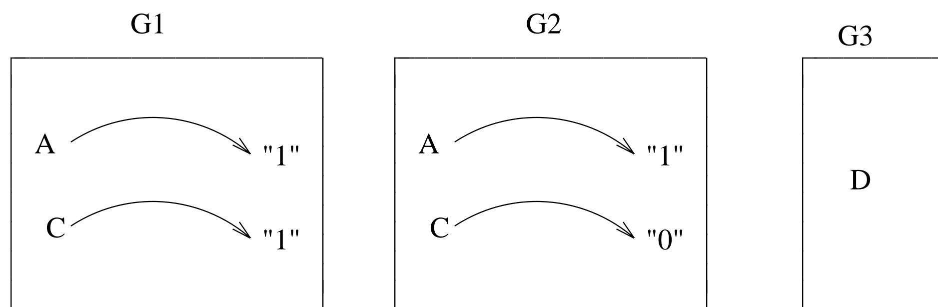

Fig. 9. A hypernode database satisfying $A \rightarrow D$ and $D \rightarrow C$ but not $A \rightarrow C$.

Theorem 1. The axiom system comprising inference rules R1-R4 is sound and complete for the class of hypernode databases.

Proof: See Appendix.

We note that the above axiom system comprising, R1-R4, was also shown to be sound and complete for FDs with respect to the class of relational databases with a single unmarked null value [LIEN82]. This implies that, within the hypernode model, we can capture the semantics of incomplete information proposed in [LIEN82] without explicitly storing the missing information in the database. These semantics fit in well with nulls of the type "value does not exist", which can be modelled by isolated attribute names (see Section I).

We next show how condition H1, which asserts the uniqueness of the defining label of every hypernode in a database HD, can be used explicitly to define the concept of key in the hypernode model. Let $\$id \in$ **AN** be a distinguished attribute such that $\$id \notin$ PRIM(HD), and assume that the HNQL program, shown in Fig. 10, *enhances* HD by adding to each hypernode, say G = (N, E), in the database an arc ($id, G). We call the resulting database an enhanced (hypernode) database; obviously the class of enhanced databases is a proper subset of the class of hypernode databases.

```
TB
  for_all X ∈ db() do
  TB
    Y := insert_node(X, $id);
    Y := insert_node(X, X);
    Y := insert_arc(X, $id, X);
  TE
TE.
```

Fig. 10. An HNQL program to enhance a hypernode database.

We now add the following two inference rules to our axiom system for HFDs.

(R5) Identity: $\forall A_i \in U, F \vdash \{\$id\} \rightarrow \{A_i\}$.

(R6) Superkey: if $F \vdash A \rightarrow \{\$id\}$, then $\forall A_i \in U, F \vdash A \rightarrow \{A_i\}$.

Theorem 2. The axiom system comprising inference rules R1-R6 is sound and complete for the class of enhanced databases.

Proof: See Appendix.

III. THE HYPERNODE MODEL AS AN UNDERLYING FORMALISM FOR HYPERTEXT

Hypertext [CONK87, NIEL90] is text that can be read nonsequentially, in contrast to traditional text, for example in book form, which has a single linear sequence defining the order in which the text is to be read. Hypertext presents several different options to readers, and the individual reader chooses a particular sequence at the time of reading. A *hypertext database* (known as a network in hypertext terminology) is a digraph (in [TOMP89] a directed hypergraph is considered) whose nodes represent units of information and whose arcs (known as links in hypertext terminology) allow the reader to navigate from an anchor node to a destination node. In the context of this paper we only consider textual units of information but, in general, hypertext

databases integrate into the system multimedia data types such as graphics, sound and video. The activity of navigating within a hypertext database by traversing links and examining the text associated with destination nodes is called *browsing*. As was pointed out in [HALA88] browsing does not provide sufficient access to a hypertext database, since the database may have a complex digraph structure rendering navigation difficult. This may cause readers to get "lost in hyperspace" [CONK87, NIEL90], i.e. readers may not know where they are and/or do not know how to get to some other position in the network. Therefore, declarative querying and searching facilities are needed in order to complement browsing. Querying can be done via the structure of the digraph using a query language (Graphlog was suggested in [CONS89]) and searching can be done by textual content using full-text retrieval techniques.

The hypernode model possesses a number of features which make it a natural candidate for being a formal model for hypertext.

Firstly, a hypernode is a digraph structure with two built-in link types. The first link type is the arc representing a referential relationship and the second link type is the encapsulated label representing a part-of relationship. Furthermore, attributes allow us to give additional semantics to nodes. In fact, hypernodes can model arbitrary complex objects. In order to support text directly we can assume that the domain of atomic values is actually a domain of textual fragments over which full-text retrieval operations are possible. In Fig. 11 we show part of a hypertext database, called PAPERS, which stores on-line papers from scientific journals. In particular, the figure shows an *overview diagram* [NIEL90] of the papers that are adjacent to PAP1 (i.e. PAP7 and PAP3) and adjacent from PAP1 (i.e. PAP11, PAP4 and PAP15); we assume that PAP1 is currently being browsed. The hypernodes encapsulated in IN1, IN2, OUT1, OUT2 and OUT3 are *annotations* of links [NIEL90]. An annotation of a link provides additional information about the link such as the name of the creator of the link, the date it was created and the subject matter of the link (see Fig. 12 for the details of the annotation OUT1). In addition to the annotation OUT1, Fig. 12 shows the hypernode PAP1, which is currently being browsed and two of its encapsulated hypernodes, AUTH1 (showing the details of one of the authors of the paper) and TEXT1 (which contains the actual text of the paper).

Secondly, the hypernode model can provide for browsing and declarative querying facilities via HNQL. HNQL can also cater for *authoring* [NIEL90] via its update facilities. Finally, within the context of the hypernode model we can reason about integrity constraints (see Section II-C) in a hypertext database. In summary we view hypertext as a promising application of the hypernode model.

IV. COMPARISON OF THE HYPERNODE MODEL TO OTHER DATA MODELS

A. Comparison to other graph-based data models

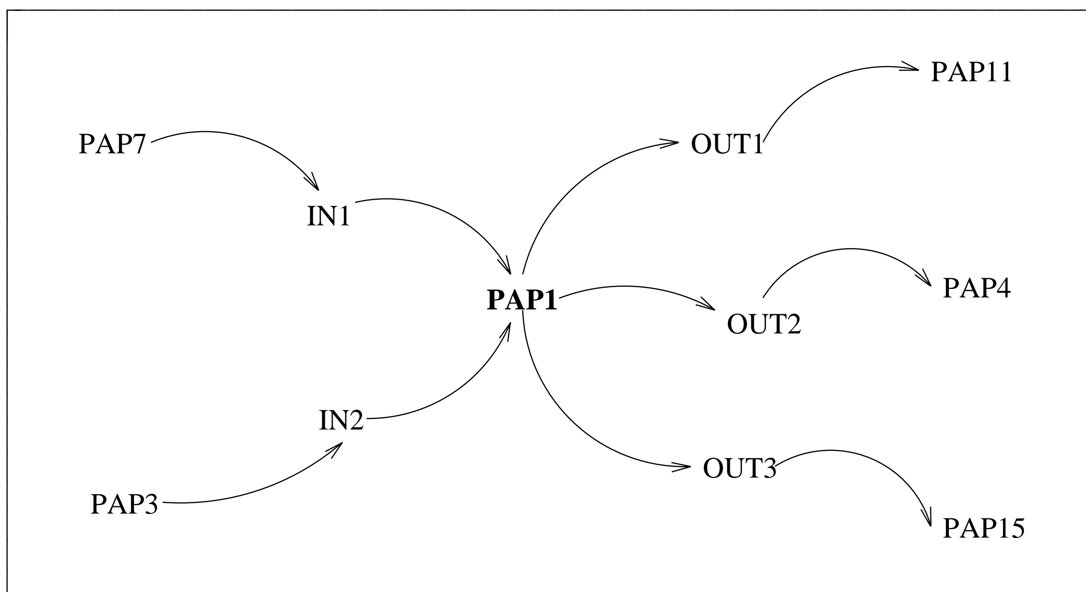

Fig. 11. Part of a hypertext database.

We briefly compare the hypernode model to other recent graph-based data models with respect to their data modelling capabilities. In particular, we deal with the Logical Data Model (LDM) [KUPE84], GOOD and G-Log [GYSS90, PARE91], and Graphlog [CONS90].

In all of the above graph-based data models the database consists of a single digraph, while a hypernode database consists of a finite set of digraphs. This unique feature of the hypernode model permits data encapsulation and the ability to represent each real-world object in the database separately.

In LDM database schemas are represented by digraphs and their instances are represented as two-column tables each of which associates entities of a particular type (which is either a primitive type, a tuple type or a set type) with their corresponding values. In the hypernode model we have a single data structure, i.e. the digraph, which as was shown in [LEVE90] has the ability to represent LDM's three types.

In GOOD, G-Log and LDM there is a separation between the database schema and the database instance, while the hypernode model, as presented herein, has no such separation, since hypernodes are untyped. This has the advantage that changes to the database can be dynamic but on the other hand it has the disadvantage that typing constraints cannot be imposed on the database.

Unlike GOOD, G-Log and Graphlog, we do not label arcs in the hypernode model. However, we can attain the same data modelling expressiveness by including arcs, which have the same label in a GOOD, G-Log or Graphlog digraph, within the arc set of a single hypernode

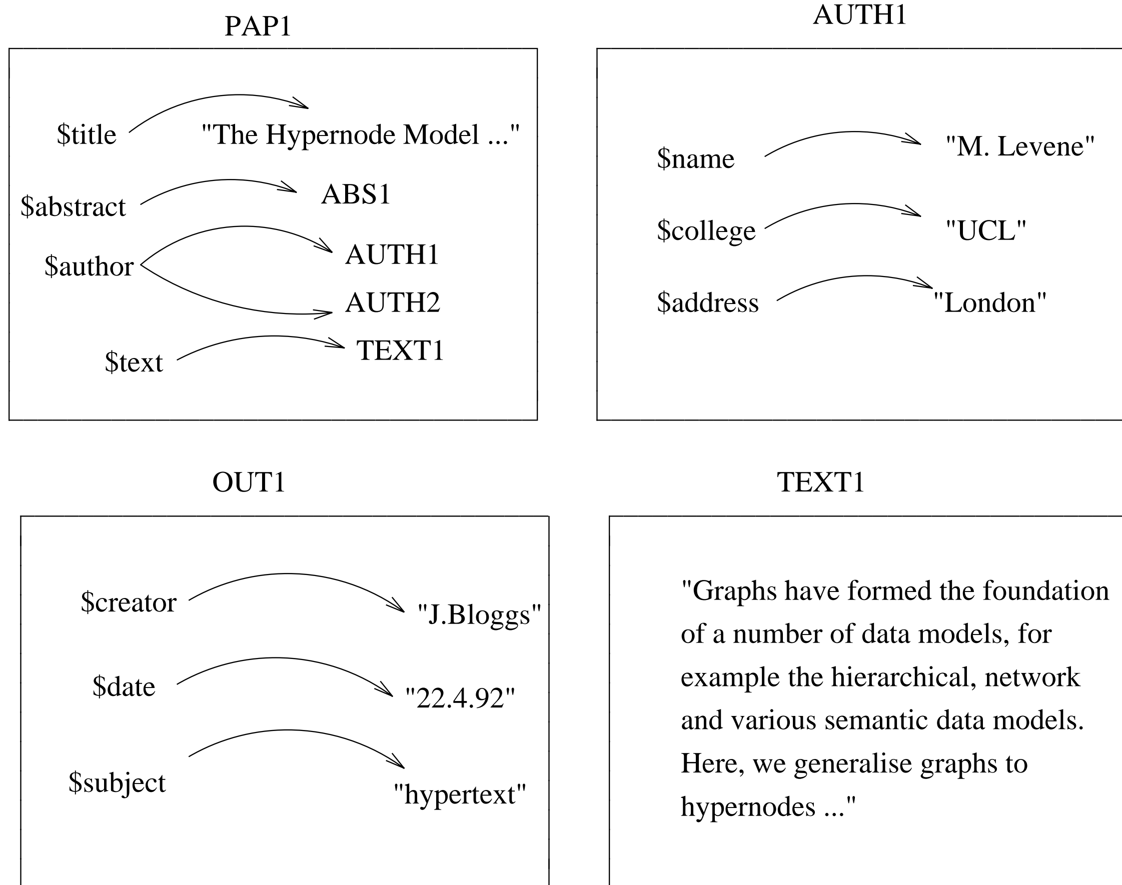

Fig. 12. Some hypernodes in the hypertext database.

whose defining label is this same label. For example, we can represent the set of labelled arcs:

Ref1 $\rightarrow^{\text{Links_to}}$ Doc1
Ref2 $\rightarrow^{\text{Links_to}}$ Doc2
Ref3 $\rightarrow^{\text{Links_to}}$ Doc3

by the hypernode:

Links_to = ({Ref1, Doc1, Ref2, Doc2, Ref3, Doc3}, {(Ref1, Doc1), (Ref2, Doc2), (Ref3, Doc3)})

We next briefly compare the hypernode model to object-oriented data models [KIM90] again with respect to their data modelling capabilities. Typically, object-oriented data models support tuple, set and list data constructors, which are used to define the type of an object. Each object has a unique object identity and belongs to only one class. The class of an object defines both its structure and its behaviour, i.e. its type and the methods it responds to.

The hypernode model supports only a single general-purpose digraph constructor, which, as was mentioned above, has the ability to represent tuple, set and list constructors. Hypernodes are

provided with object identity via their unique labels. We do not support classes in the hypernode model, since hypernodes are untyped. Furthermore, we do not support methods in the hypernode model, since we have a general-purpose database language, i.e. HNQL, which allows us to directly pose to the database any query or update definable in HNQL. As was noted in [ULLM91] such a general-purpose database language is essential in database applications such as scientific and market applications (see [TSUR91] for more details on such applications), where a large variety of queries and updates, which cannot always be planned in advance, may be posed to the database.

B. Comparison to set-based data models

Set-based data models such as the relational model [CODD70, PARER89, ULLM88] and the nested relational model [LEVE92, PARE89, THOM86, VANG88] are *value-based*. That is, tuples in relations (respectively nested relations) are identified solely by their attribute values. On the other hand, graph-based data models such as the hypernode model are *object-based*, i.e. hypernodes are identified by unique labels that serve as system-wide object identifiers.

In [ULLM88] it is argued that query and update languages for value-based data models are, in general, more declarative than those for object-based data models. For example, in [ULLM91] it is shown that the consequence of attaching object identity to each path [BUCK90] in a digraph may cause some undesirable side-effects. In particular, when trying to generate the set of all paths in a digraph we may mistakenly generate the set of all walks [BUCK90], since different walks that are induced by the same path have different object identities. If the digraph is cyclic then an infinite number of walks (i.e. objects) will be generated. Furthermore, if we are only interested in the reachability relation between the nodes of a digraph, then generating all the walks first is obviously ineffective. Although declarativeness is generally desirable, we firmly believe that navigational features supported by languages for graph-based data models such as the hypernode model are necessary in certain applications such as hypertext. Furthermore, we claim that there need not be any loss of data independence in object-based data models and thus their query and update languages need not be less declarative than those for value-based data models. We substantiate this claim with reference to our model as follows: the unique labels of hypernodes are system generated via the operator create() (they should be machine-independent to allow portability) and therefore their internal values are hidden from the users. In order to overcome this problem, unique meaningful aliases can be used at the conceptual level of the database in order to identify the defining labels of hypernodes, as we have demonstrated throughout the paper.

We close this section with a brief mention of integrity constraints in graph-based and set-based data models. In Section II-C we showed how FDs can be incorporated into the hypernode model. In order for the theory to be comprehensive we need to extend our results to include other kinds of integrity constraint such as *inclusion dependencies* [VARD88], which can be used to

enforce referential integrity constraints. In the context of the relational model there is a plethora of data dependencies which are described in [VARD88]. It would be a pity not to tap this wealth of ideas when investigating data dependencies for graph-based data models. In [LEUC91] a step has been taken in this direction by reconstructing tuple and equality generating dependencies [VARD88] in a graph-theoretic setting. In the context of an object-oriented data model PFDs mentioned in Section II-C have been generalised to *path constraints* [COBU91], which can also assert equations and typing constraints.

V. ON THE EXPRESSIVE POWER OF HNQL

A fundamental measure of the expressive power of a query and update language is the class of transformations from databases to databases (termed *computable updates*) that such a language can express. We cannot use directly the standard notion of a Turing computable mapping from strings to strings to define the said class for two reasons. Firstly, database domains are normally abstract and thus do not have a built-in ordering; this is called the *genericity* requirement (in [CHAN80] the genericity requirement is called the consistency criterion). For example, the domains **L** and **P** of the hypernode model are abstract domains and are thus uninterpreted with respect to any ordering that can be defined on them. Secondly, we may introduce non-determinism into the database language in two ways:

(1) by allowing the creation of new objects with arbitrarily chosen object identifiers (as we do by using the create() operator of HNQL defined in Section II-B); and

(2) by introducing explicit non-deterministic operators into the language (as we do via the five non-deterministic operators of HNQL also defined in Section II-B).

The non-determinism introduced by (1) is motivated by the fact that the internal values of object identifiers are hidden from the users and therefore, at least from the users' point of view, their generation should be non-deterministic. On the other hand, the non-determinism introduced by (2) is motivated by the need to answer queries such as: choose an arbitrary seat number for a passenger on a given flight or choose an arbitrary referenced paper on a specific topic from a given set of references.

As a result of the above we define two classes of computable updates: *generic computable updates* which cater for the genericity requirement and *arbitrary-order computable updates* which cater for the non-determinism introduced by (1) and (2). We show that the former class is a special case of the latter class and then investigate the expressiveness of HNQL with respect to the class of arbitrary-order computable updates.

We first introduce some useful notation. In the following we let HD be a database, π be an isomorphism from $\mathbf{L} \cup \mathbf{P}$ to ω, where ω is the set of all natural numbers, π^{-1} be the inverse of π and δ be a Turing computable mapping from strings to strings. We next define certain auxiliary

operators used in the sequel:

(1) encode(π)(HD) returns a standard encoding [GARE79], say SE, of π(HD). (We note that π preserves the adjacency of the digraphs of the hypernodes in HD.)

(2) decode(π^{-1})(SE) returns a database π^{-1}(ISE), where ISE is the result of computing the inverse of the standard encoding SE output by the encode operator.

(3) δ(SE) denotes the result of computing δ via a Turing machine that computes δ with input SE.

We observe that π is necessary in defining encode and decode, since when a set is represented by a string, the elements of the set need to be ordered [GARE79].

We are now ready to formalise the notion of a computable update.

A mapping τ, from databases to databases, is a *computable update* if there exists a Turing computable mapping δ from strings to strings and an isomorphism π such that

τ(HD) = decode(π^{-1})(δ(encode(π)(HD))).

A query and update language is *update complete* with respect to a class of computable updates if and only if it expresses all and only all the computable updates in that class.

We observe that in our context a query can be considered to be a special case of an update, since we can always put the result of a query into a distinguished hypernode and remove this hypernode from the database after the user has inspected its contents.

A. Generic Computable Updates

We now introduce the notion of a generic computable update.

A computable update τ is *generic* if it commutes with every isomorphism ρ that maps primitive nodes to primitive nodes and labels to labels, i.e. for any database, HD, $\rho(\tau(HD)) = \tau(\rho(HD))$.

We note that in [ABIT90] a more general definition of genericity is considered where a finite set C of constants (or primitive nodes in the case of the hypernode model), some of which may appear in the query or update itself, are mapped to themselves. Herein for simplicity we have assumed that C is the empty set. We further note that genericity is a consequence of the requirement that a database provide a high degree of data independence. This is due to the fact that data independence requires a computable update to be independent of the internal representation of the data. In particular if an ordering, imposed on the underlying domains at the physical level of the database, is not known at the conceptual level, then such an ordering should not affect the result

of a given update.

B. Arbitrary-Order Computable Updates

Generic computable updates do not allow us to express certain computable updates such as "choose a member from a set", since such an update is not generic. As noted in [ABIT89], this update can easily be expressed in the presence of a total ordering on the underlying domains, since we can then treat a set as an ordered list without duplicate elements. Alternatively, we can allow this sort of update to be expressed by introducing non-deterministic operators, such as the non-deterministic operators of HNQL, which permit us to choose a member from a set by introducing an arbitrary order on the members of the set (cf. the *cut* operator in Prolog [CLOC81] and the *choice* predicate in LDL [NAQV89]).

We next define a class of computable updates which takes into account the added expressiveness of the non-deterministic operators of HNQL. We then investigate the expressive power in HNQL with respect to the said class of computable updates.

A binary relation τ, from databases to databases, is an *arbitrary-order computable update* (or simply an AO computable update) if τ is a computable update up to a choice of π, which is used when computing τ.

We observe that by the definition of an AO computable update we may have (HD, HD1) $\in$ τ and (HD, HD2) $\in$ τ resulting from two different choices of π, say π_1 and π_2.

We note that if $\mathbf{L} \cup \mathbf{P}$ has a fixed natural ordering, which is always used when computing τ, then the definition of an AO computable update would degenerate to the definition of a computable update, since only one choice of π would ever be used. We further note that, in general, the definition of an AO computable update is weaker than the definition of a computable update in the sense that we do not assume that a given choice of π is a priori available when computing τ.

The following lemma shows that generic computable updates are just a special case of AO computable updates.

Lemma 3. τ is a generic computable update if and only if τ is independent of the choice of π, which is used when computing τ.

Proof: See Appendix.

If HNQL’s non-deterministic operators (including the create() operator) are to be interpreted as AO computable updates, they must behave deterministically once a choice of π is made. We now formalise this approach by defining the semantics of HNQL’s non-deterministic operators.

Let S be a set over **L** ∪ **P** (that is we make no assumption about the internal structure of S). We now define the following three auxiliary operators:

(1) *member*(S) returns an arbitrary member $s \in S$.

(2) *list*(S, π) returns the list resulting from imposing an ordering π on the set S.

(3) *first*(L) returns the first element of the list L.

We next formalise the notion of "returns an arbitrary member" by replacing the definition of *member*(S) in (1) above by

given a choice of π, *member*(S) = *first*(*list*(S, π)); otherwise *member*(S) is undefined. (α)

Thus, we have replaced the choice of an arbitrary member in the definition of *member*(S) by a choice of an arbitrary ordering imposed by a given choice of π, with the returned value of *member*(S) being the first element of the chosen ordering on S.

We can now make the assumption that when an HNQL program, say Prog, which may contain non-deterministic operators, is evaluated we can utilise (α) by making a particular choice of π prior to the evaluation of Prog. That is, when a non-deterministic HNQL operator in Prog returns an arbitrary member (which may be a node, an arc or a label) from a given set, say S, (α) is used (with the particular choice of π) in order to compute the returned value of *member*(S). From now on we call this assumption with respect to the operational semantics of HNQL's non-deterministic operators *assumption* (α). We note that the create() operator can also utilise assumption (α) by returning the least label not present in LABELS(HD′), where HD′ is the current state of HD, according to the ordering imposed by the choice of π. We illustrate the deterministic behaviour resulting from assumption (α) with a simple example. Let HD = {G = ({n_1, n_2}, ∅)} and assume that the simple HNQL program, shown in Fig. 13, is executed with respect to HD.

```
TB
  X := any_node(G);
  Y := delete_node(G, X);
TE.
```

Fig. 13. A simple HNQL program.

After the above program is executed the current state of HD is either {G = ({n_1}, ∅)} or {G = ({n_2}, ∅)}, depending on whether the choice of π induces $n_1 < n_2$ or $n_2 < n_1$.

We are now ready to state the main result of this section, which characterises the expressiveness of HNQL, given assumption (α), as an operational semantics to the non-deterministic operators of HNQL.

Theorem 4. Given assumption (α), HNQL is update complete with respect to the class of AO computable updates.

Proof: See Appendix.

In [ABIT90] a non-deterministic computable update (called a non-deterministic database transformation therein) is a binary relation τ, from databases to databases, which is generic and recursively enumerable. Although this approach is semantically "clean", it does not provide us with an operational semantics for non-determinism, since due to genericity, there is no mechanism to decide which output to choose from a query or update given a set of possible outputs. On the other hand, AO computable updates provide us with a "clean" operational semantics to non-determinism, since the choice of π in assumption (α) can be made, for example, by using the physical layout of the database. Since this layout changes over time the result of a query or an update will "appear" to the user to be non-deterministic. (For more discussion on computable updates see [ABIT89, ABIT90, CHAN80, CHAN88, HULL90, HULL91, NAQV89].)

VI. BRIDGING THE GAP BETWEEN GRAPH-BASED AND SET-BASED DATA MODELS

In this section we endeavour to bridge the gap between graph-based and set-based data models by considering a transformation from one to the other.

We first define the important notions of *copy* and *copy elimination*. Two hypernode databases are defined to be *copies* of each other if they are not equal and there exists an isomorphism from one to the other that maps primitive nodes to themselves (i.e. it is the identity mapping on **P**) and labels to labels. Intuitively, this means that the two databases are modelling exactly the same set of objects. A hypernode database *with copies* is a database such that two or more of its subsets (consisting of hypernodes) are copies of each other. Finally, the operation of *copy elimination* is the operation of replacing a database with copies by a maximal subset of this database without copies, i.e it is the operation of removing the duplicate copies. We observe that copy elimination can be performed easily in HNQL, since we can arbitrarily retain one of the copies from the duplicates and then remove the others. (For more discussion on copy elimination see [ABIT89, HULL90].)

Now, since a set-based data model can be viewed as a special case of a graph-based data model, i.e. one in which the database is such that it is always without copies, we need only consider transforming graph-based to set-based data models. This would be straightforward if graph-based data models had a built-in copy elimination operator which would be invoked on the database after each query or update is computed. We use our model to demonstrate the said transformation. In effect, we would like the hypernode model to behave like a set-based data model. This involves solving two problems. The first problem is to find a suitable set-based formalism such that the hypernode model behaves like such a formalism, and the second problem is to devise a

transformation from the hypernode model to this suitable set-based formalism.

We suggest *non-well-founded sets* [ACZE88] (also called *hypersets* [BARW91]) as a solution to the first of these two problems. Hypersets subsume well-founded sets by dropping the requirement that sets have a hierarchical structure, thus allowing us to model various kinds of circular phenomena whereby a set may contain itself. It was shown in [ACZE88] that certain systems of equations have unique solutions in the universe of hyperests. That is, a hyperset can be viewed as the unique solution to such a system of equations. This important result is called the *solution lemma*. As as a consequence of the solution lemma we can define *hyper-relations* to be the unique solutions to the set of hypernodes (which can be viewed as a set of equations whose indeterminates constitute its set of unique defining labels) in a hypernode database, HD. This interpretation will allow us to transform our model into a value-based data model thus solving our second problem. Although this solution is appealing theoretically, in practice we are faced with the problem of copy elimination, i.e. if the solution to two different defining labels is the same hyper-relation, then how is this to be detected and at what computational cost? In other words how can we test for the equality of two hyper-relations?

We now show that in the hypernode model solving this problem is at least as hard as testing for isomorphism of digraphs [BUCK90], whose complexity is, in general, an open problem (we note that the subgraph isomorphism problem is NP-complete) [GARE79]. We briefly describe two isomorphism tests, a *local* test and a *global* test, which can be used to solve copy elimination in the hypernode model.

In order to check whether two hypernodes

$G_1 = (N_1, E_1)$
$G_2 = (N_2, E_2)$

are locally isomorphic, we restrict the mapping realising the isomorphism so that primitive nodes map to themselves (i.e. it is the identity mapping on **P**), labels map to labels and the label G_1 maps to the label G_2. For example, consider the following three hypernodes:

$G_1 = (\{\$boss, G_1\}, \{(\$boss, G_1)\})$
$G_2 = (\{\$boss, G_2\}, \{(\$boss, G_2)\})$
$G_3 = (\{\$boss, G_4\}, \{(\$boss, G_4)\})$

The first hypernode is locally isomorphic to the second hypernode but not locally isomorphic to the third hypernode, since in this case G_1 would have to be mapped to both G_3 and G_4.

We observe that the above local test is, in general, not sufficient when considering the isomorphism of two hypernodes in a database, since condition H2 may be violated if one of the isomorphic hypernodes is removed from the database. For example, consider the following four hypernodes:

$G_1 = (\{\$child, G_3\}, \{(\$child, G_3)\})$

$G_2 = (\{\$child, G_4\}, \{(\$child, G_4)\})$

$G_3 = (\{\$name, "iris"\}, \{(\$name, "iris")\})$

$G_4 = (\{\$name, "robert"\}, \{(\$name, "robert")\})$

The first hypernode is locally isomorphic to the second hypernode but the third hypernode is not locally isomorphic to the fourth hypernode, since "iris" cannot map to "robert". Furthermore, if we eliminate one of the first two hypernodes, then either the third or the fourth hypernode will lose a *parent* as a result.

A global isomorphism test can now be devised by taking into account the hypernode accessibility graphs (HAGs) of the two hypernodes under consideration. In order to test whether two hypernodes, with the defining labels, G_1 and G_2, are globally isomorphic, we can use the following algorithm. First test whether the HAGs of G_1 and G_2 are isomorphic such that G_1 maps to G_2. If the answer is no, then the test fails. Otherwise, if the answer is yes, let ρ be the one-to-one mapping which verifies this isomorphism. Then test, for each node G in the node set of the HAG of G_1 whether the hypernode whose defining label is G is locally isomorphic to the hypernode whose defining label is ρ(G). If all such local tests succeed, then the global isomorphism test succeeds, otherwise it fails.

For example, the following two digraphs are, respectively, the HAGs of G_1 and G_2 shown above.

$(\{G_1, G_3\}, \{(G_1, G_3)\})$

$(\{G_2, G_4\}, \{(G_2, G_4)\})$

It can easily be verified that these HAGs are indeed isomorphic, where G_1 maps to G_2. Finally, the hypernode defined by G_1 is not globally isomorphic to the hypernode defined by G_2, since as was shown above the hypernode defined by G_3 is not locally isomorphic to the hypernode defined by G_4.

VII. CONCLUDING REMARKS AND FURTHER RESEARCH

We have presented the three components of the hypernode model in detail: its single underlying data structure, the hypernode, its query and update language, HNQL, and its integrity constraints in the form of HFDs. We have also presented hypertext as a natural application for the hypernode model, and compared the model with other graph-based data models and with set-based data models. Finally, using our model as an example, we have demonstrated that hypersets can be used to bridge the gap between graph-based and set-based data models; this is achieved at the cost of testing isomorphism of digraphs, whose complexity is, in general, an open problem. From a practical point of view, this may imply that one can bridge the gap for hierarchical databases only, i.e. when the digraphs of all hypernodes in the database are trees, since testing for

isomorphism of trees can be solved in polynomial time [GARE79].

An advantage of graph-based database formalisms, which will be important in the next generation of database systems, is that they considerably enhance the usability of complex systems [HARE88]. In particular, graph-based formalisms encourage "graphical" user interfaces.

We now list some further topics which demand more attention:

- Utilising the wealth of algorithmic graph theory in order to incorporate into HNQL special-purpose operators, which solve particular distance-related problems such as finding the shortest distance path between two nodes.
- Developing a higher level database language for the hypernode model on top of HNQL (a step in this direction is the logic-based language Hyperlog discussed in [LEVE90, POUL92]).
- Developing further our ideas of a formal model for hypertext.

We close by briefly mentioning that a first prototype of a storage manager for the hypernode model has already been implemented [TUV92]. The storage manager is a set of modules carrying out manipulation of digraphs in a persistent store. It caters for object identity and referential integrity of hypernodes, for the storage of large and dynamic hypernodes, for clustering strategies on secondary storage and for retrieval operations which utilise indexing techniques. It also supports the basic set operators of HNQL excluding its non-deterministic operators.

APPENDIX

PROOF OF THEOREM 1

Proof: It can easily be shown that the axiom system comprising R1-R4 is sound. In order to prove completeness we need to show that if $F \not\vdash A \to B$, then $F \not\models A \to B$. Equivalently for the latter, we have to exhibit a database, say EX, such that $EX \models F$ but $EX \not\models A \to B$. Let EX be the database shown in Fig. 9, where A, C and D, as before, denote pairwise disjoint sets of attributes and such that $C = U - A^+$ and $D = A^+ - A$.

We first show that $EX \models F$. Suppose to the contrary that $EX \not\models F$ and thus there exists an HFD, $V \to W \in F$, such that $EX \not\models V \to W$. It follows by the construction of EX that $V \subseteq A$ and that $\exists \$c \in (W \cap C)$ such that $\$c \notin A^+$. By R2 it follows that $F \vdash A \to W \cup A$, and by R4 it follows that $F \vdash A \to \{\$c\}$. This leads to a contradiction, since it follows that $\$c \in A^+$.

We conclude the proof by showing that $EX \not\models A \to B$. Suppose to the contrary that $EX \models A \to B$; by the construction of EX, $B \subseteq A^+$, implying that $\forall \$b \in B$, $EX \models A \to \{\$b\}$. Therefore, $\forall \$b \in B$, $F \vdash A \to \{\$b\}$ and by R3 it follows that $F \vdash A \to B$. This leads to a contradiction, since we have derived $F \vdash A \to B$. □

PROOF OF THEOREM 2

Proof: We observe that the identity inference rule, R5, is a consequence of condition H1 and thus it is sound (cf. the simple attribution rule in [WEDD92]). Furthermore, the key inference rule, R6, is a consequence of the fact that if $F \models A \rightarrow \{\$id\}$ then A is a superkey (i.e. a superset of a key) for every enhanced database satisfying F, so it is sound. From these two observations and from Theorem 1 it follows that the axiom system comprising R1-R6 is sound.

In order to prove completeness we need to show that if $F \not\vdash A \rightarrow B$, then $F \not\models A \rightarrow B$. Equivalently for the latter, we have to exhibit an enhanced database, say EXE, such that $EXE \models F$ but $EXE \not\models A \rightarrow B$. Let EXE be the enhanced database shown in Fig. 14, where A, C, D and $\{\$id\}$ denote pairwise disjoint sets of attributes and such that $C = U - (A^+ \cup \{\$id\})$ and $D = A^+ - (A \cup \{\$id\})$.

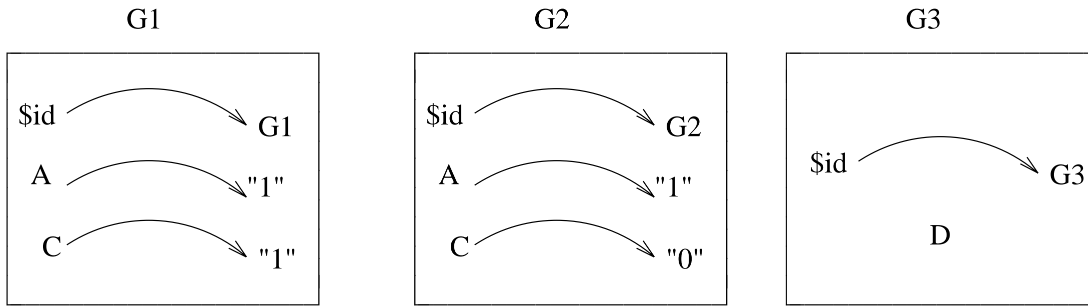

Fig. 14. An enhanced database satisfying F but not $A \rightarrow B$.

We first show that $EXE \models F$. Suppose to the contrary that $EX \not\models F$ and thus there exists an HFD, $V \rightarrow W \in F$, such that $EXE \not\models V \rightarrow W$. It follows by the construction of EXE that $V \subseteq A$ and that either $\exists \$c \in (W \cap C)$ such that $\$c \notin (A^+ - \{\$id\})$ or $\$id \in W$. In the first case the result follows as in the proof of Theorem 1, so we assume that $\$id \in W$. By R2 and R4 it follows that $F \vdash A \rightarrow \{\$id\}$. On using R6 we can deduce that $\exists \$c \notin A^+$ such that $F \vdash A \rightarrow \{\$c\}$. This leads to a contradiction, since it follows that $\$c \in A^+$.

The proof that $EXE \not\models A \rightarrow B$ is similar to that in the proof of Theorem 1; we note that by the construction of EXE, $A \neq \{\$id\}$, otherwise this would have allowed us to derive $F \vdash A \rightarrow B$ by R5. □

PROOF OF LEMMA 3

Proof: Let δ be the Turing computable mapping that is used to compute $\tau(HD)$ and let "o" denote composition.

(*If*): Since τ is independent of the choice of π which is used when computing τ we have that $\tau(HD) = decode(\pi_1^{-1})(\delta(encode(\pi_1)(HD))) = decode(\rho^{-1} \text{ o } \pi_1^{-1})(\delta(encode(\pi_1 \text{ o } \rho)(HD)))$ for any isomorphism π_1. We now have that $\rho(\tau(HD)) = \rho(decode(\rho^{-1} \text{ o } \pi_1^{-1})(\delta(encode(\pi_1 \text{ o } \rho)(HD)))) =$

decode(ρ o ρ^{-1} o π_1^{-1})(δ(encode(π_1 o ρ)(HD))) = decode(π_1^{-1})(δ(encode(π_1 o ρ)(HD))) = τ(ρ(HD)) as required.

(*Only if*): Assume that τ is not independent of the choice of π. Then there exist at least two choices of π, say π_1 and π_2, such that HD1 = decode(π_1^{-1})(δ(encode(π_1)(HD))), HD2 = decode(π_2^{-1})(δ(encode(π_2)(HD))) and HD1 $\neq$ HD2.

It follows that either τ(HD) = HD1 or τ(HD) = HD2 but not both; assume that τ(HD) = HD1. Let $\rho = \pi_1^{-1}$ o π_2. We assume without loss of generality that the isomorphism, ρ, maps primitive nodes to primitive nodes and labels to labels. Now, by the definition of ρ we have that π_1(ρ(HD)) = (π_1 o π_1^{-1} o π_2)(HD) = π_2(HD). It therefore follows that encode(π_1)(ρ(HD)) = encode(π_2)(HD) and thus also δ(encode(π_1)(ρ(HD))) = δ(encode(π_2)(HD)). Now, let SE be the result of this Turing machine computation. Hence it follows that ρ(HD1) = decode(π_1^{-1})(SE), on using the genericity requirement, since ρ(HD1) = ρ(decode(π_1^{-1})(δ(encode(π_1)(HD)))) = decode(π_1^{-1})(δ(encode(π_1)(ρ(HD)))) = decode(π_1^{-1})(SE).

Thus, since $\rho^{-1} = \pi_2^{-1}$ o π_1, we have that HD1 = decode(π_2^{-1} o π_1 o π_1^{-1})(SE) = decode(π_2^{-1})(SE). This concludes the proof, since it is also the case that HD2 = decode(π_2^{-1})(SE) and thus HD1 = HD2, leading to a contradiction. □

PROOF OF THEOREM 4

Proof: Let HNQL^{det} denote the subset of HNQL without its non-deterministic operators. It can easily be shown that all the updates expressed by HNQL^{det} are in fact generic computable updates. Thus by Lemma 3 all the updates expressed by HNQL^{det} are also AO computable updates. Now, by assumption (α), HNQL's non-deterministic operators become AO computable updates. Thus, since the choice of π is made prior to the evaluation of a given HNQL program, it follows by structural induction on the constructs of HNQL programs that HNQL expresses only AO computable updates.

In order to conclude the proof we need to show that HNQL can express all the AO computable updates. Now, let τ be any AO computable update. It is immediate from the definition of the non-deterministic operators of HNQL that a given choice of π, say π', can be generated in HNQL. We observe that for any current state of HD, say HD′, π' need only be defined for PRIM(HD′) $\cup$ LABELS(HD′).

Now, let δ be the Turing computable mapping that is used to compute τ(HD) with π' being the isomorphism used in the definition of a computable update. We next need to show that decode(π'^{-1})(δ(encode(π')(HD))) can be simulated in HNQL. Firstly, encode(π')(HD) can be simulated in HNQL by using a standard encoding scheme as in [GARE79]. An example of the result of encoding the hypernode, G = ({n_1, n_2}, {(n_1, n_2)}), ignoring π', for the sake of brevity, is shown in Fig. 15. Secondly, δ(SE) can be simulated in HNQL, since HNQL is capable of simulating the working of any Turing machine; this is realised by using certain results given in

ENCODE

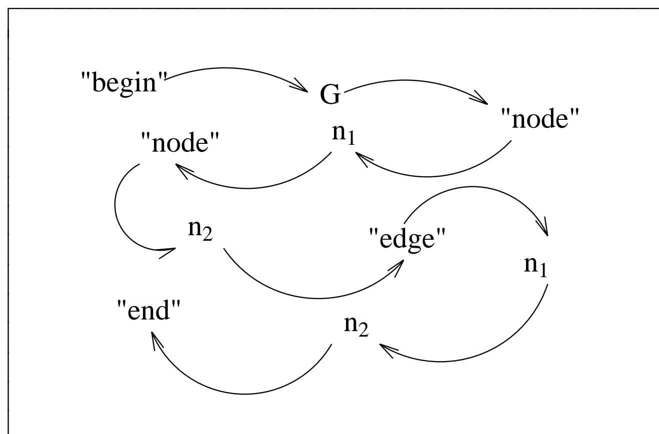

Fig. 15. An example of encoding a hypernode.

[LEVE90]. Finally, decode(π'^{-1})(δ(SE)) can be simulated in HNQL by defining in HNQL the inverse mapping of the encode operator. We leave the remaining technical details of the proof to the reader; the techniques used are similar to those used in [ABIT89, ABIT90, CHAN80] to prove update completeness. □

ACKNOWLEDGEMENT

The work of Mark Levene is funded by grant GR/G26662 of the U.K. Science and Engineering Research Council. He would like to thank Alexandra Poulovassilis for their joint work on the hypernode model.

REFERENCES

[ABIT89] S. Abiteboul and P.C. Kanellakis, "Object identity as a query language primitive", in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Portland, Ore., pp. 159-173, 1989.

[ABIT90] S. Abiteboul and V. Vianu, "Procedural languages for database queries and updates", *Journal of Computer and System Sciences*, vol. 41, pp. 181-229, 1990.

[ACZE88] P. Aczel, *Non-well-founded Sets*. Lecture Notes no. 14. Stanford, Ca.: Center for the Study of Language and Information (CSLI), 1988.

[BARW91] J. Barwise and L. Moss, "Hypersets", *The Mathematical Intelligencer*, vol. 13, pp. 31-41, 1991.

[BUCK90] F. Buckley and F. Harary, *Distance in Graphs*. Redwood City, Ca.: Addison-Wesley, 1990

[CHAN80] A.K. Chandra and D. Harel, "Computable queries for relational data bases", *Journal of Computer and System Sciences*, vol. 21, pp. 156-178, 1980.

[CHAN88] A.K. Chandra, "Theory of database queries", in *Proceedings of ACM Symposium on Principles of Database Systems*, Austin, Texas, pp. 1-10, 1988.

[CHEN76] P.P-S. Chen, "The Entity-Relationship model - towards a unified view of data", *ACM Transactions on Database Systems*, vol. 1, pp. 9-36, 1976.

[CLOC81] W.F. Clocksin and C.S. Mellish, *Programming in Prolog*. Heidelberg, Germany: Springer-Verlag, 1981.

[COBU91] N. Coburn and G.E. Weddell, "Path constraints for graph-based data models: Towards a unified theory of typing constraints, equations and functional dependencies ", in *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, pp. 312-331, 1991.

[CODD70] E.F. Codd, "A relational model of data for large shared data banks", *Communications of the ACM*, vol. 13, pp. 377-387, 1970.

[CODD79] E.F. Codd, "Extending the database relational model to capture more meaning", *ACM Transactions on Database Systems*, vol. 4, pp. 397-434, 1979.

[CONK87] J. Conklin, "Hypertext : An introduction and survey", *IEEE Computer*, vol. 20, pp. 17-41, 1987.

[CONS89] M.P. Consens and A.O. Mendelzon, "Expressing structural hypertext queries in Graphlog", in *Proceedings of the ACM Conference on Hypertext*, Pittsburgh, Pa., pp. 269-292, 1989.

[CONS90] M.P. Consens and A.O. Mendelzon, "Graphlog : a visual formalism for real life recursion", in *Proceedings of ACM Symposium on Principles of Database Systems*, Nashville, Tennessee, pp. 404-416, 1990.

[DERO89] S.J. DeRose, "Expanding the notion of links", in *Proceedings of the ACM Conference on Hypertext*, Pittsburgh, Pa., pp. 249-257, 1989.

[GARE79] R.G. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman and Co., 1979.

[GYSS90] M. Gyssens, J. Paredaens and D. Van Gucht, "A graph-oriented object database model", in *Proceedings of ACM Symposium on Principles of Database Systems*, Nashville, Tennessee, pp. 417-424, 1990.

[HALA88] F.G. Halasz, "Reflections on Notecards: Seven issues for the next generation of Hypermedia systems", *Communications of the ACM*, vol. 31, pp. 836-852, 1988.

[HARE88] D. Harel, "On visual formalisms", *Communications of the ACM*, vol. 31, pp. 514-530, 1988.

[HULL90] R. Hull and M. Yoshikawa, "ILOG: Declarative creation and manipulation of object identifiers", in *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia, pp. 455-468, 1990.

[HULL91] R. Hull and J. Su, "On the expressive power of database queries with intermediate types", *Journal of Computer and System Sciences*, vol. 43, pp. 219-267, 1991.

[JENS85] K. Jensen and N. Wirth, *Pascal User Manual and Report*. Heidelberg, Germany: Springer-Verlag, 1985.

[KIM90] W. Kim, *Introduction to Object-Oriented Databases*. Cambridge, Ma.: MIT Press, 1990

[KUPE84] G.M. Kuper and M.Y. Vardi, "A new approach to database logic", in *Proceedings of ACM Symposium on Principles of Database Systems*, Waterloo, Ontario, pp. 86-96, 1984.

[LIEN82] Y.E. Lien, "On the equivalence of database models", *Journal of the ACM*, vol. 2, pp. 333-362, 1982.

[LEVE92] M. Levene, *The Nested Universal Relation Database Model*. Lecture Notes in Computer Science, vol. 595. Heidelberg, Germany: Springer-Verlag, 1992.

[LEVE90] M. Levene and A. Poulovassilis, "The hypernode model and its associated query language", in *Proceedings of the Jerusalem Conference on Information Technology*, Jerusalem, Israel, pp. 520-530, 1990.

[LEVE91] M. Levene and A. Poulovassilis, "An object-oriented data model formalised through hypergraphs", *Data and Knowledge Engineering*, vol. 6, pp. 205-224, 1991.

[LEUC91] J. Leuchner, L. Miller and G. Slutzki, "Agreement graph dependencies.", *Bulletin of the EATCS*, vol. 45, pp. 202-217, 1991.

[MAIE84] D. Maier, J.D. Ullman and M.Y. Vardi, "On the foundations of the universal relation model", *ACM Transactions on Database Systems*, vol. 9, pp. 283-308, 1984.

[NAQV89] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases*. Rockville, Md.: Computer Science Press, 1989.

[NIEL90] J. Nielsen, *Hypertext & Hypermedia*. San Diego, Ca.: Academic Press, 1990.

[PARE89] J. Paredaens, P. De Bra, M. Gyssens and D. Van Gucht, *The Structure of the Relational Database Model*. Heidelberg, Germany: Springer-Verlag, 1989.

[PARE91] J. Paredaens, P. Peelman and L. Tanca, "G-Log: A declarative graphical query language", in *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, pp. 108-128, 1991.

[POUL92] A. Poulovassilis and M. Levene, "A nested-graph model for the representation and manipulation of complex objects", *ACM Transactions on Information Systems*, in press, 1992.

[RICH91] J. Richardson and P.Schwarz, "Aspects: Extending objects to support multiple, independent roles", in *Proceedings of the ACM SIGMOD International Conference on the Management of Data*, Denver, Colorado, pp. 298-307, 1991.

[TOMP89] F.W. Tompa, "A data model for flexible hypertext database systems", *ACM Transactions on Information Systems*, vol. 7, pp. 85-100, 1989.

[THOM86] S.J. Thomas and P.C. Fischer, "Nested relational structures", in *Advances in Computing Research* **3**, P.C. Kanellakis and F. Preparata Eds. JAI Press, Greenwich, pp. 269-307, 1986.

[TSUR91] S. Tsur, "Deductive databases in action", in *Proceedings of ACM Symposium on Principles of Database Systems*, Denver, Colorado, pp. 142-153, 1991.

[TUV92] E. Tuv, A. Poulovassilis and M. Levene, "A storage manager for the hypernode model", to appear in *Proceedings of 10th British National Conference on Databases*, Aberdeen, U.K., 1992.

[ULLM88] J.D. Ullman, *Principles of Database and Knowledge-Base Systems, Volume I.* Rockville, Md.: Computer Science Press, 1988.

[ULLM91] J.D. Ullman, "A comparison between deductive and object-oriented database systems", in *Proceedings of the International Conference on Deductive and Object-Oriented Databases*, Munich, Germany, pp. 263-277, 1991.

[VANG88] D. Van Gucht and P.C. Fischer, "Multilevel nested relational structures", *Journal of Computer and System Sciences*, vol. 36, pp. 77-105, 1988.

[VARD88] M.Y. Vardi, "Fundamentals of dependency theory", in *Trends in Theoretical Computer Science*, Chapter 5, E. Borger Ed. Computer Science Press, Rockville, Md., pp. 171-224, 1988.

[WEDD92] G.E. Weddell, "Reasoning about functional dependencies generalized for semantic data models", *ACM Transactions on Database Systems*, vol. 17, pp. 32-64, 1992.