A Nested-Graph Model for the Representation and Manipulation of Complex Objects

ALEXANDRA POULOVASSILIS
King's College, London
and
MARK LEVENE
University College, London

Three recent trends in database research are object-oriented and deductive databases and graph-based user interfaces. We draw these trends together in a data model we call the Hypernode Model. The single data structure of this model is the *hypernode*, a graph whose nodes can themselves be graphs. Hypernodes are typed, and types, too, are nested graphs. We give the theoretical foundations of hypernodes and types, and we show that type checking is tractable. We show also how conventional type-forming operators can be simulated by our graph types, including cyclic types. The Hypernode Model comes equipped with a rule-based query language called Hyperlog, which is complete with respect to computation and update. We define the operational semantics of Hyperlog and show that the evaluation of Hyperlog programs is intractable in the general case—we identify cases when evaluation can be performed efficiently. We discuss also the use of Hyperlog for supporting database browsing, an essential feature of Hypertext databases. We compare our work with other graph-based data models—unlike previous graph-based models, the Hypernode Model provides inherent support for data abstraction via its nesting of graphs. Finally, we briefly discuss the implementation of a DBMS based on the Hypernode Model.

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures—*graphs*; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.3 [**Database Management**]: Languages—*query languages*

General Terms: Design, Languages

Additional Key Words and Phrases: Complex object, nested graph, object store, rule-based query and update language, types

The Hypernode Project is supported financially by the U.K. Science and Engineering Research Council (grant number GR/G26662).
Authors' addresses: A. Poulovassilis, Department of Computer Science, King's College London, Strand, London, WC2R 2LS, U.K. email: alex@uk.ac.kcl.dcs; M. Levene, Department of Computer Science, University College London, Gower Street, London, WC1E 6BT, U.K. email: m.levene@uk.ac.ucl.cs

1. INTRODUCTION

Recent database research has focused on deductive [Ceri et al. 1990; Naqvi and Tsur 1989] and object-oriented [Beeri 1989; Kim 1990] databases.[1] Deductive databases extend the relational data model with rule-based computation. Rules enable the derivation of further, intentional tuples from the stored, extensional tuples. These derived tuples can be used purely for querying purposes or can be inserted into the database. Conversely, object-oriented databases start off with a semantic data model [Griffith 1982; Hull and King 1987; Shipman 1981], which typically supports object identity, inheritance, and complex objects, and extend it with features such as methods and encapsulation from object-oriented programming [Shriver and Wegner 1987; Wong and Lochovsky 1989].

Thus, deductive and object-oriented databases are largely complementary. The former support extensionally and intentionally defined relations, but not fundamental data abstraction concepts such as classification, identification, inheritance, and encapsulation. Conversely, the latter do support these abstraction concepts but do not support relations naturally. Hence, recent research has aimed at integrating the two paradigms. The integration has generally taken the route of extending logic-based deductive database languages with features such as object identity, sets, functions, methods, and inheritance [Abiteboul and Kanellakis 1989; Abiteboul and Vianu 1987; 1988]. In contrast, in this paper we report on a *graph-based* approach to such an integration.

Our use of graphs has two key advantages: First, graphs are formally defined, well-understood structures; second, it is widely accepted that graph-based formalisms considerably enhance the usability of complex systems [Harel 1988]. Graphs have been used in conjunction with a number of conventional data models, for example, the hierarchical and network models [Ullman 1988], the entity-relationship model [Chen 1976] and a recent extension thereof for complex objects [Parent and Spaccapietra 1989], and various semantic data models [Griffith 1982; Hull and King 1987; Shipman 1981]. Graphs or hypergraphs [Berge 1973] have also been used more recently [Consens and Mendelzon 1990; Gyssens et al. 1990; Kuper and Vardi 1984; Levene and Poulovassilis 1991; Tompa 1989; Watters and Shepherd 1990] as a data-modeling tool in their own right. We give a comparison between this recent work and our own approach in Section 4.

Directed graphs have also been the foundation of Hypertext databases [Conklin 1987; Tompa 1989]. Such databases are graphs consisting of nodes which refer to units of stored information (typically text) and of named links. Each link connects two nodes, the "source" and the "destination." Links are traversed either forward (from source to destination) or backward (from destination to source). The process of traversing named links and examining the text associated with nodes is called *browsing*. Typically, a simple query facility consisting of string-based search is provided which can be used to

[1] See also *Proceedings of the International Conference on Deductive and Object-Oriented Databases* (1989).

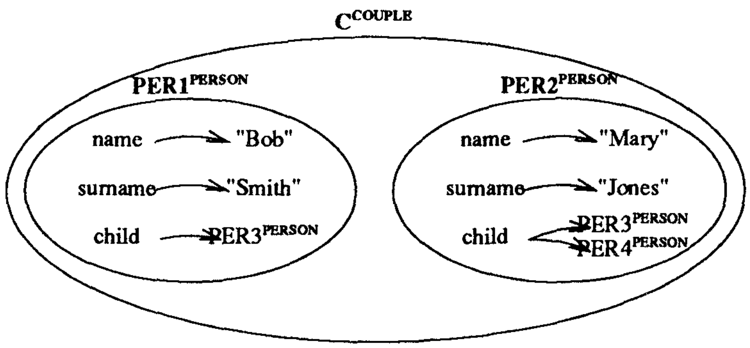


Fig. 1. An example hypernode.

identify an initial set of nodes prior to browsing. A further feature of Hypertext is the dynamic creation of new nodes and links.

Motivated by the previous research outlined above, we have developed a graph-based data model called the Hypernode Model, which supports object identity and arbitrarily complex objects and which is well-suited to the implementation of Hypertext databases. In contrast to other graph-based models, we use nested, possibly recursively defined, graphs termed *hypernodes*. A hypernode is a pair (N, E) of nodes and directed edges such that the nodes can themselves be hypernodes. Thus, unlike other graph-based models, the Hypernode Model provides inherent support for the *nesting* of information. The labels of hypernodes are unique and serve as *object identifiers*. We illustrate a hypernode in Figure 1. It represents a couple, C, consisting of two people, PER1 and PER2, whose children are nested within further hypernodes. In Figure 2 we show the children of person PER1, which would become visible if we "exposed" the hypernodes labeled PER3 and PER4. We observe from these figures that hypernodes differ from *hypergraphs* in that they generalize nodes to hypernodes as opposed to generalizing edges to hyperedges.

We note that the labels C and PER1–PER4 in Figure 1 are superscripted with the tags COUPLE and PERSON, respectively. As we explain in the sequel, these tags indicate the *types* of their associated hypernodes. Types give us a means of defining database schemas and of enforcing constraints on the structure and content of hypernodes. Types, too, are represented by nested graphs and can be queried and updated using the same formalism as for hypernodes. Also, we note the use of the node $\text{none}^{\text{PERSON}}$ in Figure 2—it denotes "not present."

The Hypernode Model comes equipped with a computationally powerful declarative language called Hyperlog. The model and language share features with both deductive and object-oriented databases. In common with other deductive database languages, Hyperlog is rule based and supports derivations and database updates. In common with object-oriented databases the Hypernode Model supports arbitrarily complex objects and the data abstraction concepts of classification (via types), identification (via unique labels),

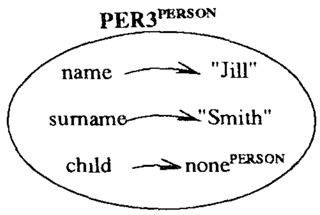


Fig. 2. Further hypernodes.

and encapsulation (via the nesting of graphs). In Levene and Poulovassilis [1991] we showed how structural inheritance is also supported naturally by nested-graph structures (in that paper, we used nested hypergraphs, but our treatment is equally applicable to the simpler nested graphs of the Hypernode Model). In Levene and Poulovassilis [1990] we also showed how methods can be supported as parametrized Hyperlog programs.

The Hypernode Model supports the main features of Hypertext databases: strings of arbitrary length are supported as a primitive type, and so unstructured text can be represented. Such text can be a node of a hypernode which is itself encapsulated within a number of further hypernodes—hence text can be shared. Sets of text fragments are easily represented—as the nodes of a hypernode. Annotated links can be represented by a hypernode with a single incoming edge from the source node and a single outgoing edge to the destination node; this hypernode can encapsulate the annotation information, for example, the actual link label, a description of the semantics of the link, the creator of the link, and its date of creation. The nesting of hypernodes is an abstraction tool which greatly facilitates the design and browsing of densely connected database graphs and which is unique to our model. Finally, Hyperlog can support both database browsing and general-purpose declarative querying. The latter facility can be used to create contexts for browsing.

We first introduced the Hypernode Model in Levene and Poulovassilis [1990]. Here we expand on that work in several directions, including expressiveness of representation and computation, efficiency of inference, support of Hypertext, and implementation issues. We describe recent work in extending the model to include types and extending Hyperlog to perform deletions as well as insertions. The outline of this paper is as follows. In Section 2 we discuss the fundamentals of the Hypernode Model, namely, hypernodes and types. We discuss representational expressiveness and type-checking complexity. In Section 3 we give the syntax and semantics of Hyperlog. We discuss the complexity of evaluating Hyperlog programs and their computational and update expressiveness. We show also how Hyperlog can be used for database browsing. In Section 4 we compare our work with other graph-based languages and models. In Section 5 we briefly describe a prototype implementation. We conclude in Section 6.

2. THE HYPERNODE MODEL

In this section we discuss the fundamentals of our model, namely, hypernodes and types. We define hypernodes and repositories for them in Section

2.1. We define types and type repositories in Section 2.2, where we examine also the efficiency of type checking. In Section 2.3 we illustrate the use of types via an extended example based on a flights bookings database. Finally, in Section 2.4 we discuss the representational expressiveness of our model.

2.1 Hypernodes and Hypernode Repositories

In this section we introduce the underlying data structure of the hypernode model, namely, the *hypernode*. We define a *hypernode repository* to be a set of graph-defining equations, and we define *hypernodes* to be the values assigned to the indeterminates when such a set of equations is solved.

We begin by recalling the definition of a directed graph—a directed graph is an ordered pair (N, E), where N is a finite set of nodes and $E \subseteq (N \times N)$ is a finite set of directed edges. For simplicity, we use the terms "graph" and "directed graph" interchangeably, similarly for the terms "edge" and "directed edge." We use also the notation $n_1 \rightarrow n_2$ interchangeably with (n_1, n_2) for edges. For the purposes of the Hypernode Model we need two disjoint sets of constants, a finite set of *primitive nodes*, **P**, and a countably infinite set of *labels*, **L**. We assume that the set **P** includes alphanumeric strings. Other elements of **P** are denoted by identifiers which start with a lower-case letter. Elements of **L** are denoted by identifiers which start with an upper-case letter.

The graphs of the Hypernode Model are defined by equations of the form

$$G = (N, E)$$

where $G \in \mathbf{L}$ and where (N, E) is a graph such that $N \subseteq (\mathbf{P} \cup \mathbf{L})$. We term such equations *hypernode equations*. Examples are the following, where P1, P2, N1, N2 are labels and name, spouse, title, "Ms," "Mr," "A," "B," "Floyd," and "Tring" are primitive nodes:

P1 = ({name, spouse, N1, P2}, {name → N1, spouse → P2})
P2 = ({name, spouse, N2, P1}, {name → N2, spouse → P1})
N1 = ({title, initial, surname, "Ms", "A", "Floyd"}, {title → "Ms", initial → "A", surname → "Floyd"})
N2 = ({title, initial, surname, "Mr", "B", "Tring"}, {title → "Mr", initial → "B", surname → "Tring"})

A *hypernode repository* (or simply a repository) is a finite set of hypernode equations satisfying the following two conditions:

(H1) no two equations have the same left-hand side;

(H2) for any label appearing in the right-hand side of an equation, there exists an equation with that label on its left-hand side.

Given a hypernode repository, HR, we denote by LABELS(HR) the set of labels appearing in the equations of HR and by PRIM(HR) the set of primitive nodes appearing in the equations of HR.

For example, the four equations above satisfy the criteria for a hypernode repository. We note that condition H1 above corresponds to the *entity integrity* requirement of Codd [1979] since each equation can be viewed as representing a real-world entity. Similarly, condition H2 corresponds to the *referential-integrity* requirement of Codd [1979] since it requires that only existing entities be referenced.

Hypernode repositories can be viewed as storing a set of graphs which may reference other graphs via their labels. Alternatively, since hypernode repositories are just sets of equations, we would like them to have a unique solution for the indeterminates (i.e., for the labels $G \in \mathbf{L}$) in some well-defined domain. This domain cannot be the universe of well-founded sets since hypernode equations may be cyclicly defined (for example, the equations defining P1 and P2 above). However, we can appeal to Aczel's theory of *non-well-founded sets* [Aczel 1988] to solve hypernode repositories. Non-well-founded sets subsume well-founded sets by including circular sets, i.e., sets that may contain themselves. It is shown in Aczel [1988] that a set of set-defining equations (of which a hypernode repository is a special case) has a unique solution in the universe of non-well-founded sets.

Thus, a hypernode repository HR has a unique solution in the universe of non-well-founded sets. This solution assigns to each label G on the left-hand side of an equation a non-well-founded set. We term such a set a *hypernode* and denote it by $\mathrm{HYP}_{\mathrm{HR}}(G)$, or simply HYP($G$) if HR is understood from the context. The hypernode HYP(G) is an ordered pair (N, E), where N is a set of primitive nodes and further hypernodes and $E \subseteq (N \times N)$ (we note that any ordered pair (a, b) can be viewed as the set {a, {a, b}}). For example, given a hypernode repository consisting of the four equations for P1, P2, N1, and N2 above, we have (ignoring the node sets of the graphs for simplicity):

HYP(N1) = {title →“Ms”, initial →“A”, surname →“Floyd”}
HYP(N2) = {title →“Mr”, initial →“B”, surname →“Tring”}
HYP(P1) = {name → HYP(N1), spouse → HYP(P2)}
= {name → {title →“Ms”, initial →“A”, surname →“Floyd”},
spouse → {name → HYP(N2), spouse → HYP(P1)}}
= {name → {title →“Ms”, initial →“A”, surname →“Floyd”},
spouse → {name → {title →“Mr”, initial →“B”, surname →“Tring”},
spouse → {. . .}}}
HYP(P2) = {name → HYP(N2), spouse → HYP(P1)}
= {name → {title →“Mr”, initial →“B”, surname →“Tring”},
spouse → {name → HYP(N1), spouse → HYP(P2)}}
= {name → {title →“Mr”, initial →“B”, surname →“Tring”},
spouse → {name → {title →“Ms”, initial →“A”, surname →“Floyd”},
spouse → {. . .}}}

We note that the sets HYP(N1) and HYP(N2) are well-founded ones while the sets HYP(P1) and HYP(P2) are non-well-founded ones since they contain themselves.

2.2 Types and Type Repositories

In this and the next two sections we extend our model to incorporate types, which are also graphs. We define *type equations*, *type repositories*, and *types* by analogy to hypernode equations, hypernode repositories, and hypernodes. We define what it means for a hypernode to be of a particular type, and we show that testing a hypernode repository for well typedness can be performed in polynomial time with respect to the size of the repository. In Section 2.3 we illustrate types via an extended example based on a flights bookings database.

For the purpose of defining types, we assume the availability of two disjoint sets of constants: a finite set of primitive types, **TP**, such that every primitive node $n \in \mathbf{P}$ is of some unique primitive type $T \in \mathbf{TP}$, and a countably infinite set of *type labels*, **TL**, such that every label in a hypernode repository is tagged by a unique type label $T \in \mathbf{TL}$ (c.f., the types of object identifiers in object-oriented databases). By analogy to primitive nodes and labels, we distinguish between primitive types and type labels by using identifiers which start with a lower-case letter for the former and identifiers which start with an upper-case letter for the latter. We assume that for every type T, primitive or otherwise, there is a distinguished primitive node, none^T, denoting "not present." As we will see below, this node is used to model missing or incomplete information. Finally, we assume that the set of primitive types includes the type "string."

Types are defined by means of equations of the form

$$T = (M, F)$$

where $T \in \mathbf{TL}$, and (M, F) is a graph such that $M \subseteq (\mathbf{TP} \cup \mathbf{TL})$. We call such equations *type equations*. A *type repository*, TR, is a finite set of type equations satisfying conditions H1 and H2 for hypernode repositories in Section 2.1. Again, we can appeal to the theory of non-well-founded sets to solve type repositories, i.e., to assign values to the $T \in \mathbf{TL}$ from the universe of non-well-founded sets. We call such values *types* and denote them by $\mathrm{HYP}_{\mathrm{TR}}(T)$, or $\mathrm{HYP}(T)$ when TR is understood from context. These values take the form of a pair (M, F), where M is a set of primitive types and further types, and $F \subseteq (M \times M)$. We make also the reasonable assumption that primitive nodes and labels are distinct from primitive types and type labels, i.e., $(\mathbf{P} \cup \mathbf{L}) \cap (\mathbf{TP} \cup \mathbf{TL}) = \varnothing$. Thus, there is no overlap between the data (hypernodes) and the metadata (types), and the hypernode and type repositories can be merged into one repository. As well as a uniform storage of data and metadata, this means that the metadata can be queried and updated using the same formalism as the data, namely Hyperlog.

Typings of hypernodes are defined recursively as follows. Given a hypernode (N, E) and a type (M, F) we say that (N, E) is of type (M, F) if there exists a homomorphism $\phi : N \to M$ which preserves the types and which satisfies the following conditions:

(T1) if $n \in N$, then $\phi(n) \in M$;

(T2) if $(n_1, n_2) \in E$ then $(\phi(n_1), \phi(n_2)) \in F$;

(T3) if $m \in M$ then $\exists n \in N$ such that $m = \phi(n)$;

(T4) if $(m_1, m_2) \in F$ then $\exists (n_1, n_2) \in E$ such that $m_1 = \phi(n_1)$ and $m_2 = \phi(n_2)$.

Conditions T1 and T2 stipulate that a hypernode must contain only nodes and edges which conform to the nodes and edges of the intended type, while conditions T3 and T4 stipulate that a hypernode must contain *at least one* *instance* of every node and edge in its intended type. These last two conditions are not restrictive since the primitive nodes none^T can be used in place of missing information.

Typings of individual hypernodes are generalized to typings of hypernode repositories as follows. A hypernode repository, HR, is *well typed* with respect to a type repository, TR, if for every label G^T in LABELS(HR), $\mathrm{HYP}_{\mathrm{HR}}(G^T)$ is of type $\mathrm{HYP}_{\mathrm{TR}}(T)$. The following theorem states that testing a hypernode repository for well typedness is tractable. The result follows by observing that to test a hypernode repository for well typedness, we can fix the homomorphism ϕ to map primitive nodes and labels to their types and then check the criteria T1–T4 above for each equation of HR. If there are m equations in HR to be checked and a maximum of n nodes and e edges in the right-hand side of any hypernode or type equation, the test is achieved in a time proportional to mn^2e^2.

THEOREM 1. *Testing whether a hypernode repository, HR, is well typed with respect to a type repository, TR, can be performed in a time polynomial in the number of equations in HR and the maximum size of individual equations.*

2.3 The Flights Bookings Database Example

To illustrate types we now consider a database that stores information about bookings of flights by passengers. The schema of our database is specified by the type FLIGHT_BOOKINGS_SCHEMA in Figure 3. From now on we use equation-based and pictorial representations of graphs interchangeably. We omit also the type tags of labels if these are understood from context. FLIGHT_BOOKINGS_SCHEMA contains six further types:

(i) RF, shown in Figure 4, which represents ROUTEs and the FLIGHTs that fly them (each route is followed by a number of flights).

(ii) PT, shown in Figure 5, which represents PASSENGERs and their TICKETs (each passenger has bought one or more tickets).

(iii) TC, shown in Figure 6, which represents TICKETs and their COUPONs (each ticket consists of a number of coupons). We notice the sharing of the graph TICKET by the graphs TC and PT.

(iv) FC, shown in Figure 7, which represents FLIGHTs and their COUPONs (each flight is booked by a number of coupons).

(v) AIRLINES and AIRPORTS, shown in Figure 8, which contain the known airlines and airports, respectively.

The remaining types needed to specify fully the FLIGHT_BOOKINGS_SCHEMA are AIRLINE and AIRPORT, shown in Figure 9, TIME, DATE, FARE, and NAME, shown in Figure 10, and finally NAT, shown in Figure 11.

We can make several observations from the above types:

(i) Edges in types can be used to represent *attributes*, for example, flight_no_att → NAT in ROUTE. We adopt the convention that primitive types which end in "_att" represent the attribute names. There is

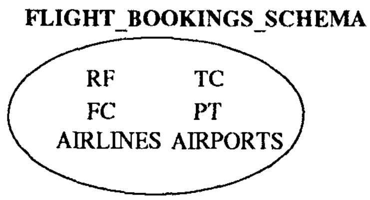


Fig. 3. The example schema.

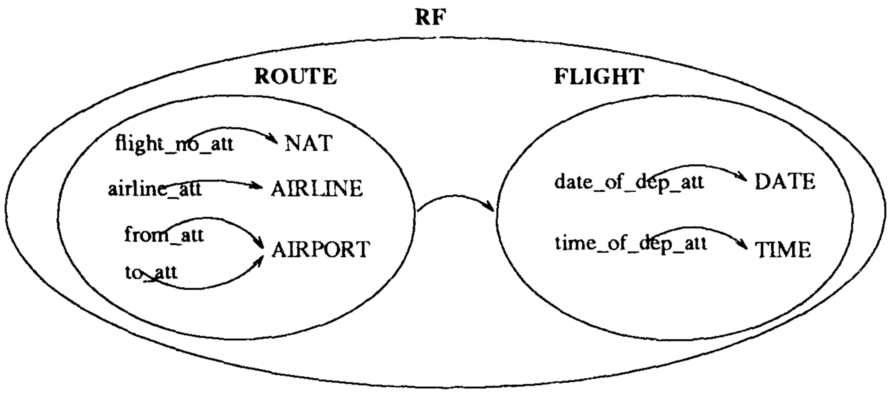


Fig. 4. Routes and flights that fly them.

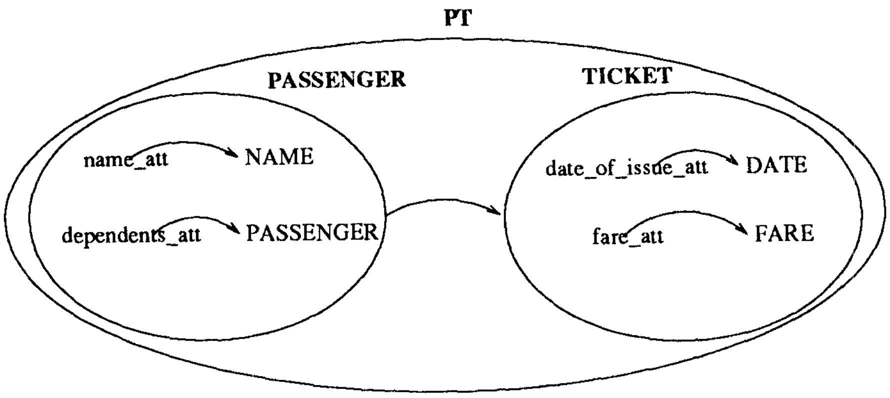


Fig. 5. Passengers and their tickets.

one primitive node of each such primitive type, and it is this node which appears in instances. For example, the primitive nodes flight_no and airline appearing in Figure 12 are assumed to be of type flight_no_att and airline_att, respectively. In practice, the user can introduce new primitive types into the set **TP** at any time and can populate these types by introducing new primitive nodes into the set **P**.

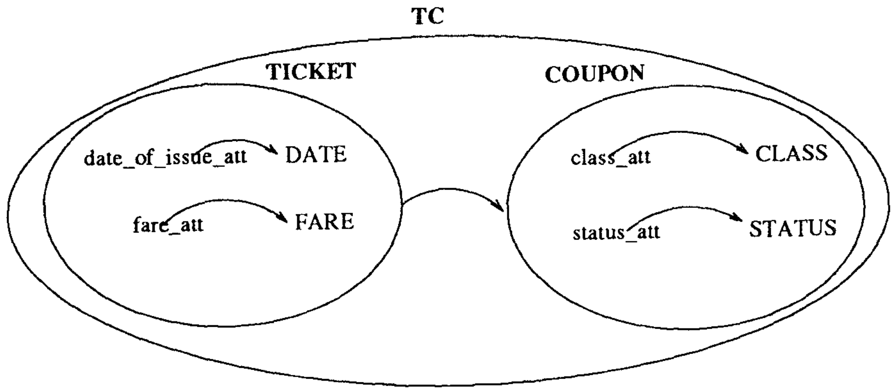


Fig. 6. Tickets and their coupons.

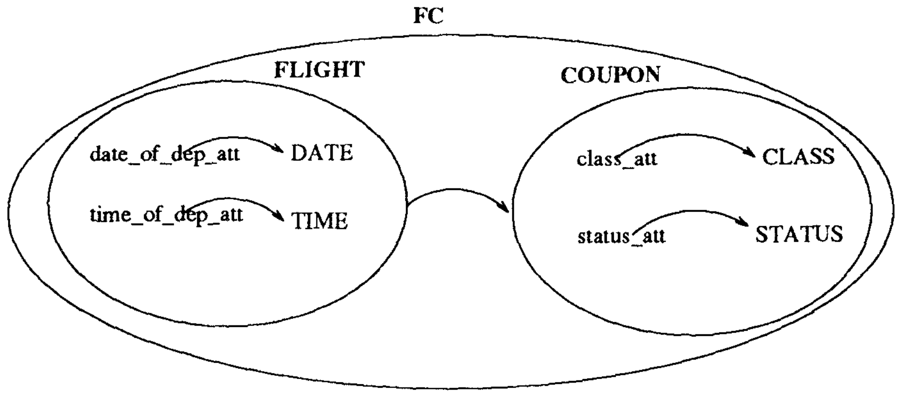
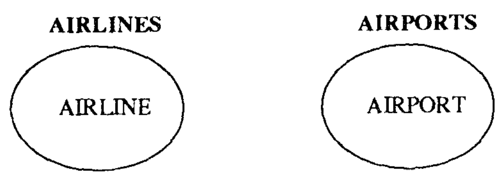


Fig. 7. Flights and their coupons.

Fig. 8. The set types AIRLINES and AIRPORTS.



(ii) Edges in types can also be used to represent binary *relationships*, for example, ROUTE → FLIGHT in RF. In general, these relationships are many-to-many due to the fact that instances are defined as being homomorphic to their type. However, cardinality constraints can be enforced within update programs—we give an example in Section 3.3 below.

(iii) It is possible to define *recursive types*, for example, PASSENGER, whose dependents are also of type PASSENGER, and NAT, which contains itself.

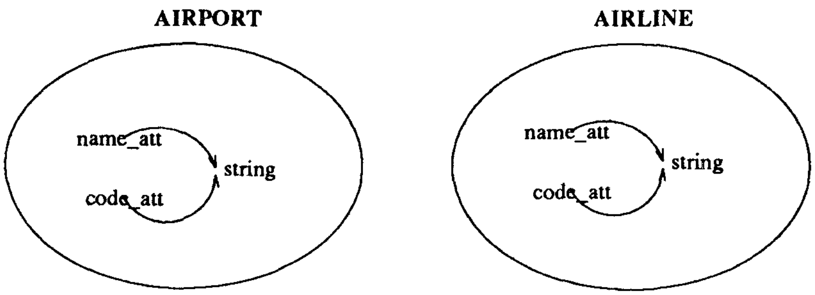


Fig. 9. The types AIRPORT and AIRLINE.

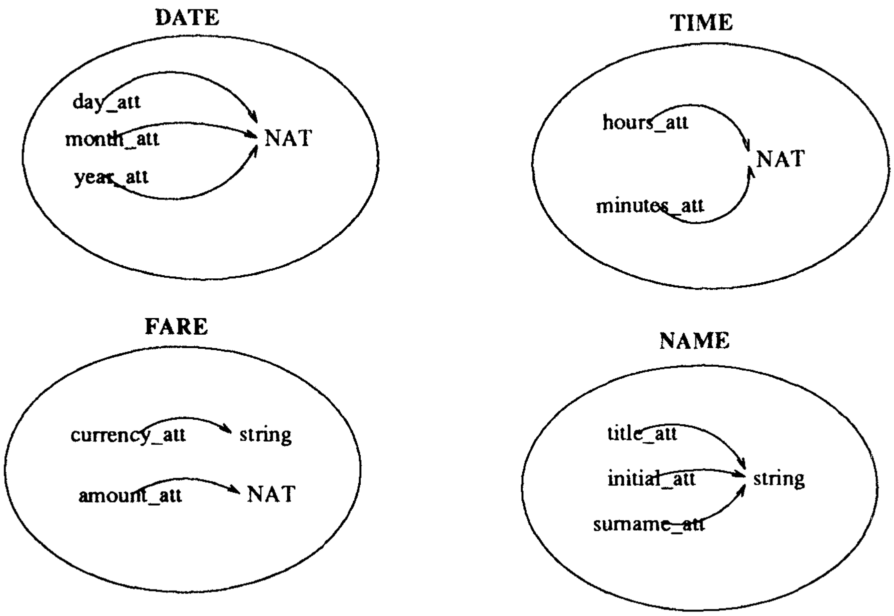


Fig. 10. The types DATE, FARE, TIME and NAME.

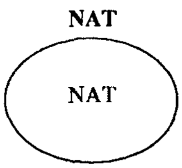


Fig. 11. The natural numbers type.

(iv) The type NAT is used to represent the *natural numbers* in the Hypernode Model; 0 is represented by a hypernode which contains the primitive node none^{NAT}, and successive natural numbers are successive nestings of 0 (see Figure 13). We describe how calculations are performed with these numbers in Section 3.6.

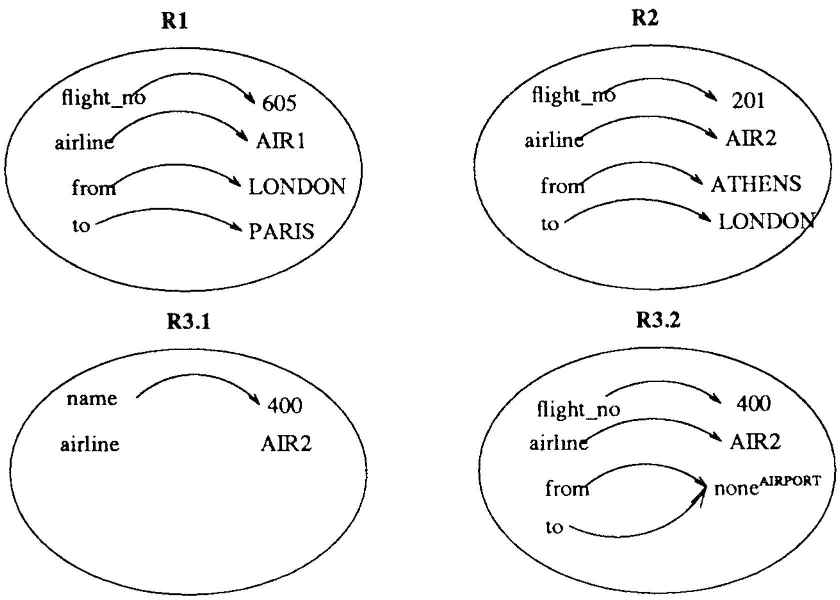


Fig 12. The routes R1, R2 and R3.

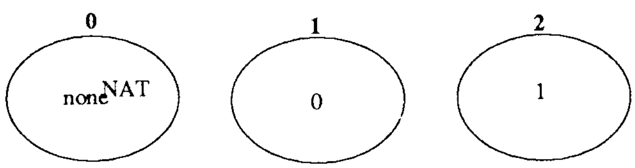


Fig. 13. The first three natural numbers.

(v) Each hypernode of type FLIGHT_BOOKINGS_SCHEMA will be a Flight Bookings database. A typical instance will have one node of each of the types RF, PT, TC, FC, AIRLINES, AIRPORTS, representing the ROUTE-FLIGHTs, PASSENGER-TICKETs, TICKET-COUPONs, and FLIGHT-COUPONs relations, and also the airlines and airports.

We note from FLIGHT_BOOKINGS_SCHEMA that schema design using the Hypernode Model is comparable with the Entity-Relationship (ER) approach [9]. There are, however, two fundamental differences between the two modeling approaches which should be stressed. First, our types can directly model complex objects, which may be hierarchical or cyclic, while these cannot be modeled directly using ER diagrams. Second, our types can encapsulate further types, for example, FLIGHT_BOOKINGS_SCHEMA encapsulates RF, TC, FC, PT, AIRLINES, and AIRPORTS, while TICKET encapsulates DATE and FARE. Such encapsulation encourages a stepwise schema design, and, in cases where the schema is large or has many interconnections, it renders the schema much easier to display and comprehend.

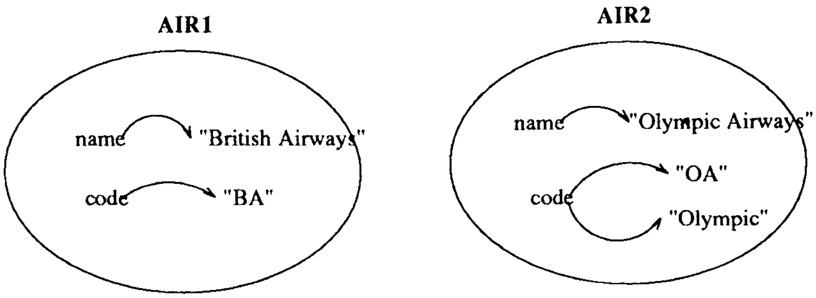


Fig. 14. Two airlines.

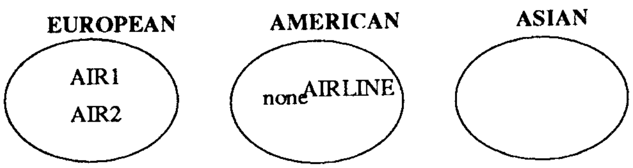


Fig. 15. The sets of airlines, EUROPEAN and AMERICAN and the incorrectly typed ASIAN.

We now illustrate some specific instances of the above types. In Figure 12 we show four hypernodes, R1, R2 and two versions of R3. We note that R1 and R2 are of type ROUTE, while the first version of R3 is not since conditions T1, T2, T3 and T4 of Section 2.2 are all violated. R3.1 can be amended to be of the type ROUTE by replacing name by flight_no, adding an edge from airline to AIR2, and specifying edges for the attributes from_att and to_att, resulting in R3.2. In Figure 13 we show the first three natural numbers. In Figure 14 we show the hypernodes AIR1 and AIR2, which are both of type AIRLINE. We note that AIR2 has two codes. In Figure 15 we show the hypernodes EUROPEAN, AMERICAN, and ASIAN. EUROPEAN and AMERICAN are both of type AIRLINES, while ASIAN violates condition T3 of this type—it can be corrected by adding to it the primitive node $\text{none}^{\text{AIRLINE}}$. In Figure 16 we show the hypernodes PT1 and PT2 of type PT. We note that the dependents can be nested to any finite depth. Finally, in Figure 17 we show the hypernodes N1 and N2 of type NAME, F1 of type FARE, and D1 of type DATE. The enforcement of meaningful dates can either be achieved by defining appropriate primitive types for day, month, and year or via the update programs.

2.4 Expressiveness of Representation

Types and type checking constitute a powerful data-modeling and integrity-checking tool since they allow database schemas to be represented and enforced. Also, storage-level optimizations can be carried out based on type information. The Hypernode Model is *type complete* in the sense that the only allowed type-forming operator (graph definition) can be applied arbitrarily

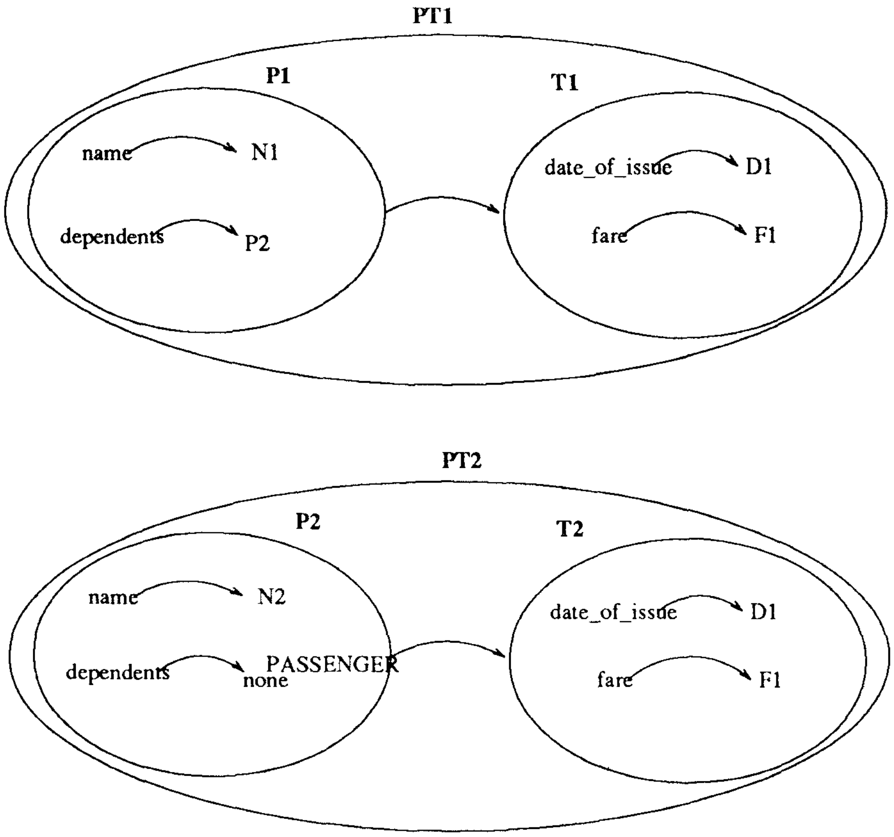


Fig. 16. Two instances of the passenger-ticket relationship.

many times. Also, cyclic types, such as PASSENGER in Figure 5 and cyclic hypernodes, such as P1 and P2 of Section 2.1, are supported. Bearing these two points in mind, we show below how some conventional type-forming operators can be simulated using graph definition.

Given a type T, we can represent a *set* type $S = \text{set}(T)$ by

$$S = (\{T\}, \emptyset)$$

Hypernodes of type S contain one or more nodes of type T, and the empty set of type S can be represented by a hypernode, $\text{EMPTY}^T = (\{\text{none}^T\}, \emptyset)$. The types AIRLINES and AIRPORTS in Figure 8 and the instances EUROPEAN and AMERICAN in Figure 15 illustrate set types.

Given types $T_1, T_2, \ldots, T_n$ and attribute names $A_1, A_2, \ldots, A_n$ (such as flight_no_att, airline_att, from_att, and to_att) we can represent a *record* type $T = [A_1{:}T_1, A_2{:}T_2, \ldots, A_n{:}T_n]$ as

$$T = (\{A_1, A_2, \ldots, A_n, T_1, T_2, \ldots, T_n\}, \{A_1 \to T_1, A_2 \to T_2, \ldots, A_n \to T_n\})$$

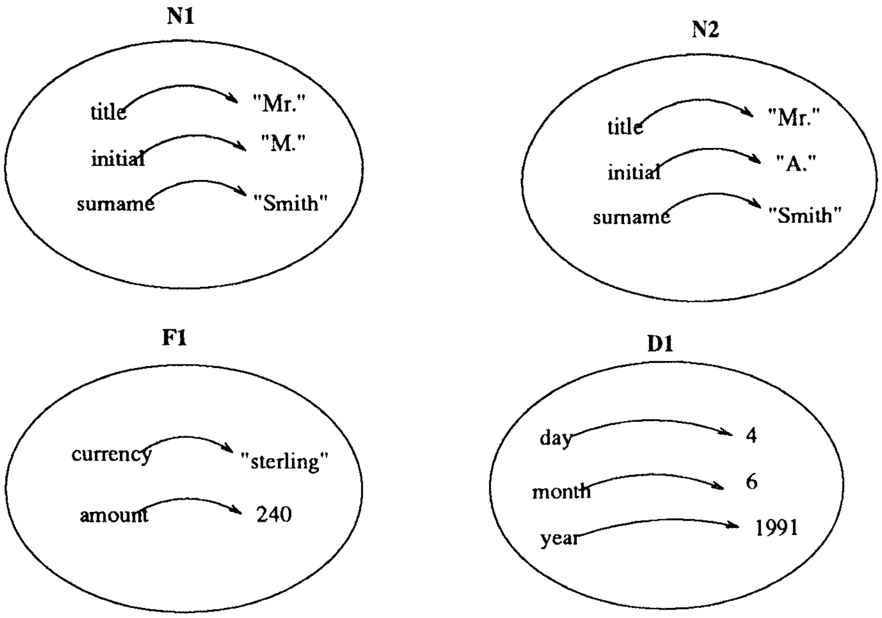


Fig. 17. The names N1 and N2, the fare F1, and the date D1.

The types ROUTE and FLIGHT of Figure 4 and TIME, DATE, FARE, and NAME shown in Figure 10 illustrate record types. We note that a record type T is a bipartite graph. We note also that this construction differs from the usual idea of a record since the attributes A_i can be multivalued, i.e., there may be more than one edge emanating from each A_i in an instance of type T. The enforcement of single values can be encoded in the update programs in Hyperlog.

Given a record type T, we can represent a *relation* type as set(T). The component types of a record type may themselves be record types, and so *nested* relations [Schek and Scholl 1986] can also be represented. The types FLIGHT of Figure 4 and TICKET of Figure 5 illustrate nested-relation types.

Given types T_1 and T_2, we can represent a *mapping* type $T = T_1 \to T_2$ as

$$T = (\{T_1, T_2\}, \{T_1 \to T_2\})$$

The types, RF, PT, TC, and FC shown in Figures 4–7 illustrate mapping types. A further example is the type NAT_TO_NAT = ({NAT}, {NAT → NAT}). Example instances of NAT_TO_NAT are the identity function ID, which maps each natural number to itself, and the mapping GREATER, which maps each natural number to all smaller natural numbers:

$$\begin{aligned} \text{ID} &= (\{0, 1, 2, \ldots, \text{MAXNUM}\}, \{0 \to 0, 1 \to 1, 2 \to 2, \ldots, \text{MAXNUM} \to \text{MAXNUM}\}) \\ \text{GREATER} &= (\{0, 1, 2, \ldots, \text{MAXNUM}\}, \{1 \to 0, 2 \to 1, 2 \to 0, \ldots, \text{MAXNUM} \to 2, \text{MAXNUM} \to 1, \text{MAXNUM} \to 0\}) \end{aligned}$$

We note that partial mappings can be represented without violating type correctness (so long as there is at least one edge in the mapping). For example, in the mapping GREATER, the element 0 is not the source of any edge, and the maximal number MAXNUM is not the sink of any edge.

Finally, given types $T_1, T_2, \ldots, T_n$ we can represent a *tuple* type $T = T_1 \times T_2 \times \cdots \times T_n$ as a record type $[I_1{:}T_1,\ I_2{:}T_2, \ldots, I_n{:}T_n]$ where the attribute types $I_1,\ I_2, \ldots, I_n$ contain the primitive constants first, second, $\ldots$, nth, respectively.

In general then, given types $T_1, T_2, \ldots, T_n$, the type T defined by

$$T = \left(\{T_1, T_2, \ldots, T_n\}, \{T_{\iota_1} \to T_{\iota_2} \cdots T_{\iota_{r-1}} \to T_{\iota_r}\}\right)$$

has instances which are *heterogeneous* sets of isolated nodes (arising from the T_ι which do not participate in any edge in T) and edges (arising from the edges of T).

3. MANIPULATION OF HYPERNODES

In this section we introduce Hyperlog, a declarative query and update language for the Hypernode Model. Hyperlog programs consist of sets of rules. The body of a rule consists of a number of graphs, called *queries*, which may contain variables and which act as templates to be matched against the equations in the hypernode repository. The head of a rule is also a query and indicates the updates (if any) to be undertaken for each match of the graphs in the body. The evaluation of a program comprises a repeated matching of its set of rules against the hypernode repository until no more updates can be inferred. In Section 3.1 below we describe the syntax of Hyperlog. In Section 3.2 we define the matching of queries in rule bodies against a hypernode repository, and in Section 3.3 we describe the inference of updates from queries in the heads of rules. In Section 3.4 we define the operational semantics of a Hyperlog program via a fixpoint operator. We then address efficiency and expressiveness issues of Hyperlog in Sections 3.5 and 3.6; in 3.5 we address the efficiency of inference and in 3.6 computational and update expressiveness. We conclude in Section 3.7 with a brief discussion on how database browsing can be supported by Hyperlog.

We have chosen a rule-based language for the Hypernode Model for two main reasons. First, the high-level, declarative nature of the language blends in well with the graph-based data model. Second, the language is very expressive: as we will see below, it is in fact complete with respect to computation and database update. As a consequence, programs which are frequently invoked can be optimized by being built-in without compromising the semantics of the language. Candidates for optimization are the arithmetic functions and the database browsing functions.

3.1 Syntax of Hyperlog

For the purposes of Hyperlog, we assume that a countably infinite set of variables, **V**, is available. We denote elements of **V** by upper-case identifiers from the end of the alphabet. We assume that the set of variables **V** and the set of labels $\mathbf{L}$ are disjoint. We assume also that all variables are typed, that is, superscripted with a type $T \in \mathbf{TP} \cup \mathbf{TL}$. However, we often omit these superscripts if they are understood from context.

Each Hyperlog rule has a, possibly empty, set of graphs in its body and a single graph in its head. We call these graphs *queries*. A query may have a variable as its label and may have variables in its node set. Also, its nodes and edges may be *negated* (meaning "absent," intuitively). More formally, a query is an equation of the form

$$Q = (N, E)$$

where $Q \in \mathbf{L} \cup \mathbf{V}$ and where (N, E) is a graph such that:

(i) $N \subseteq (\mathbf{P} \cup \mathbf{L} \cup \mathbf{V})$.

(ii) N is the disjoint union of two sets, $N+$ and $N-$. $N+$ contains "positive" nodes, and $N-$ contains "negative" nodes.

(iii) E is the disjoint union of two sets, $E+$ and $E-$. $E+$ contains "positive" edges, and $E-$ contains "negative" edges.

(iv) $(n_1, n_2) \in (E+ \cup E-)$ implies $n_1, n_2 \in N+$.

Condition (iv) restricts all edges to be between positive nodes: clearly, a positive edge containing a negative (i.e., absent) node is impossible; also, since no edge can contain a negative node, negative edges containing negative nodes are meaningless.

For simplicity, we denote a node $n \in N-$ as $\neg n$ and an edge $n_1 \to n_2 \in E-$ as $n_1 \not\to n_2$. Three examples of queries are

$$\mathrm{X}^{\mathrm{ROUTE}} = (\{\mathrm{flight\_no}, \mathrm{Y}^{\mathrm{NAT}}, \neg \mathrm{AIR1}^{\mathrm{AIRLINE}}\}, \{\mathrm{flight\_no} \to \mathrm{Y}^{\mathrm{NAT}}\})$$

which, informally, finds the route and flight number for routes not with airline AIR1,

$$\mathrm{R2}^{\mathrm{ROUTE}} = (\{\mathrm{flight\_no}, 301^{\mathrm{NAT}}\}, \{\mathrm{flight\_no} \to 301^{\mathrm{NAT}}\})$$

which, informally, checks whether route R2 has flight number 301, and

$$\mathrm{GREATER}^{\mathrm{NAT\_TO\_NAT}} = (\{10, \mathrm{X}^{\mathrm{NAT}}\}, \{10 \not\to \mathrm{X}\})$$

which finds all the numbers greater or equal to 10, using the GREATER mapping of Section 2.4.

A *Hyperlog program* is a finite set of rules, a *rule* being an expression of the form

$$q_0 \leftarrow q_1, q_2, \ldots, q_n$$

where $n \geq 0$, and $q_0, q_1, \ldots, q_n$ are queries. For example, we give below a program (ignoring the node sets of graphs for simplicity) which generates all transitive dependents of passengers and places this information into the mapping TRANS_DEPS from PASSENGERs to PASSENGERs:

$$\mathrm{TRANS\_DEPS} = \{\mathrm{Y} \to \mathrm{X}\} \leftarrow \mathrm{Y}^{\mathrm{PASSENGER}} = \{\mathrm{dependents} \to \mathrm{X}^{\mathrm{PASSENGER}}\}$$

$$\mathrm{TRANS\_DEPS} = \{\mathrm{Y} \to \mathrm{X}\} \leftarrow \mathrm{TRANS\_DEPS}^{\mathrm{PASS\_TO\_PASS}} = \{\mathrm{Y}^{\mathrm{PASSENGER}} \to \mathrm{Z}^{\mathrm{PASSENGER}}\}, \mathrm{Z} = \{\mathrm{dependents} \to \mathrm{X}^{\mathrm{PASSENGER}}\}$$

A Hyperlog program, P, can be represented as a labeled graph $P = (N, E)$ as follows. For each rule

$$q_0 \leftarrow q_1, q_2, \ldots, q_n$$

in P add to the node set N the two graphs q_0 and $R_{\text{body}} = (\{q_1, q_2, \ldots, q_n\}, \emptyset)$, and add to the edge set E the edge $R_{\text{body}} \rightarrow q_0$. We assume that the labels P and R_{body} are drawn from a set of program and rule names, **PROG**, whose members are distinct from the set of hypernode labels, **L**, and the set of type labels, **TL**. For example, the program above is represented by the graph shown in Figure 18, where DEPS_PROG, BODY1 and BODY2 are unique identifiers drawn from **PROG** (in subsequent figures of programs we often dispense with the outer program label).

We note from Figure 18 that rule heads can be shared between rules in the graphical representation of programs. In Figure 23 we give an example where rule bodies are also shared. The semantics of a shared rule head are those of *disjunction*, the head being inferred if any of the bodies are true. Conversely, the semantics of a shared rule body are those of *conjunction*, all the rule heads being inferred if the rule body is true. We finally note that our graphical representations of programs are *not* hypernodes: they are not typed, and the graphs encapsulated within them are not required to have unique labels (for example, two graphs have label TRANS_DEPS in Figure 18).

3.2 Queries in Rule Bodies

The queries in the bodies of rules act as *templates* which are matched against the equations in the hypernode repository. Before defining this matching process, we need the concept of a substitution of variables by constants of the appropriate type. A *substitution*, θ, is a set of assignments $\{X_1^{T_1}/C_1^{T_1}, X_2^{T_2}/C_2^{T_2}, \ldots, X_n^{T_n}/C_n^{T_n}\}$, where each X_i is a distinct variable in **V** and where each C_i is a distinct element in $\mathbf{L} \cup \mathbf{P}$ of the same type, T_i, as X_i. The *application* of a substitution θ to a query $Q = (N, E)$ is the equation $Q\theta = (N, E)\theta$ resulting from the substitution of each X_i in the left-hand side and right-hand side of the query by C_i. Given a hypernode repository, HR, and a query, $Q = (N+ \cup N-, E+ \cup E-)$, a *match* for the query with respect to the repository is a substitution, θ, for all the variables in the query by constants drawn from LABELS(HR) $\cup$ PRIM(HR) such that there exists an equation $Q\theta = (N', E') \in$ HR satisfying

(i) $\forall n \in N+, n\theta \in N'$.
(ii) $\forall n \in N-, n\theta \notin N'$.
(iii) $\forall e \in E+, e\theta \in E'$.
(iv) $\forall e \in E-, e\theta \notin E'$.

We can extend this definition to a set of queries $\{q_1, q_2, \ldots, q_n\}$: a substitution θ is a match for this set of queries if it is a match for each query q_i taken separately. We note that in the above definition we are assuming a Herbrand Universe and the Closed World Assumption [Reiter 1978]. This allows us to

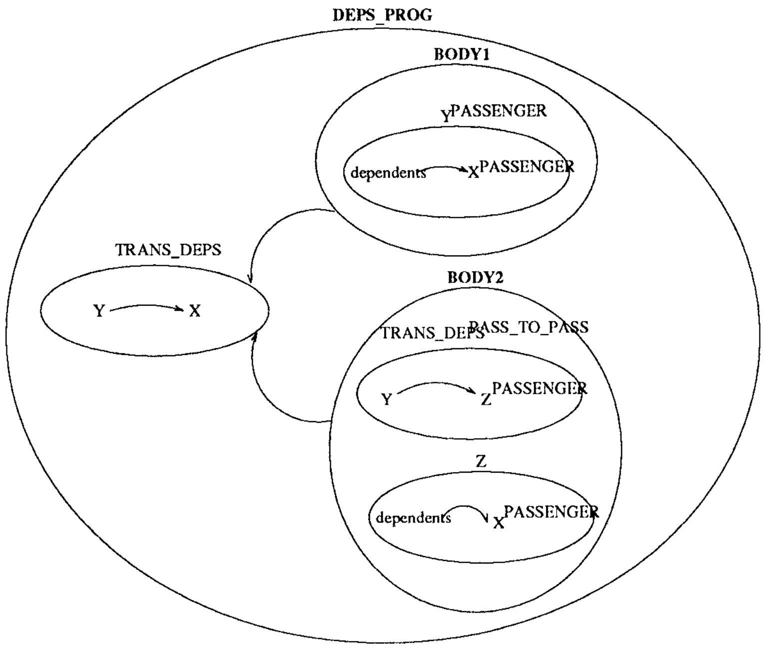


Fig. 18. Program to generate all dependents.

infer the negation of a node or edge in the absence of a positive match, e.g., other nonmonotonic formalisms [Przymusinska and Przymusinski 1990].

For example, given a hypernode repository containing the following routes

R1 = ({flight_no, 605, airline, AIR1, from, to...}, {flight_no → 605, airline → AIR1...})
R2 = ({flight_no, 301, airline, AIR2, from, to...}, {flight_no → 301, airline → AIR2...})
R3 = ({flight_no, 400, airline, AIR1, from, to...}, {flight_no → 400, airline → AIR1...})

the sets of possible matches for the four queries

$$\begin{aligned}
X^{ROUTE} &= (\{\text{flight_no}, \text{airline}, T^{NAT}, \text{AIR1}\}, \{\text{flight_no} \rightarrow Y, \text{airline} \rightarrow \text{AIR1}\}) \\
R2^{ROUTE} &= (\{\text{flight_no}, 301\}, \{\text{flight_no} \rightarrow 301\}) \\
R2^{ROUTE} &= (\{\text{flight_no}, 302\}, \{\text{flight_no} \rightarrow 302\}) \\
X^{ROUTE} &= (\{\neg\text{AIR1}\}, \emptyset)
\end{aligned}$$

are {{X/R1, Y/605}, {X/R3, Y/400}}, {{ }}, { }, and {{X/R2}}, respectively.

3.3 Queries in Rule Heads

The query in the head of a rule indicates the updates to be undertaken for each match of the queries in the body of the rule. A rule, R, in a program, P, may thus *modify* some of the equations in the hypernode repository by adding or deleting nodes and edges in their right-hand side according to positive or negative nodes and edges in the head of R.

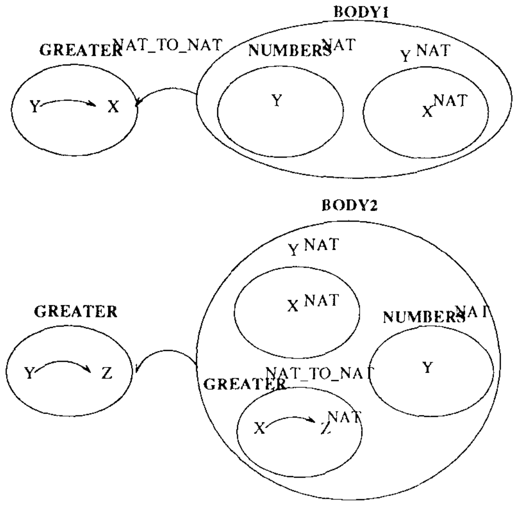


Fig. 19. Program to generate the GREATER relation.

Furthermore, there may be variables appearing in the head of R which do not appear in its body—we denote the set of such variables by NEW_R. In this case, if the head of R does not match any existing equation in the repository, a set of *new* equations is generated, one for each positive variable in NEW_R. The labels on the left-hand sides of these new equations are hitherto unused in the hypernode repository and in the program P and are chosen nondeterministically. Also, if P consists of a number of rules, $R_1, \ldots, R_r$, the sets of new labels generated for the sets of variables $\text{NEW}_{R_1} \ldots \text{NEW}_{R_r}$ are pairwise disjoint. Clearly, new labels may be left as dangling references after the execution of the program P. Thus, we relax condition H2 of our definition of hypernode repositories in Section 2.1 to assume an equation

$$G^T = \text{null}(T)$$

for any dangling label G^T, where the *null graph* null(T) is defined as follows for any type $T = (\{T_1, T_2, \ldots, T_n\}, \{T_{\iota_1} \to T_{\iota_2}, \ldots\})$:

$$\text{null}(T) = (\{\text{none}^{T_1}, \text{none}^{T_2}, \ldots \text{none}^{T_n}\}, \{\text{none}^{T_{\iota_1}} \to \text{none}^{T_{\iota_2}}, \ldots\})$$

We now illustrate some hypernode programs. The program DEPS_PROG in Figure 18 generates all transitive dependents of passengers and places this information into the mapping TRANS_DEPS. The program in Figure 19 generates the GREATER relation between natural numbers. It assumes that all the natural numbers are contained in the node set of a distinguished hypernode with the label NUMBERS. The program in Figure 20 places into a RESULT hypernode the passengers who are paying a fare of more than $200 on some ticket. The program in Figure 21 adds passenger P3 to the dependents of passenger P1, deleting any null value that might be there if P3 is the first recorded dependent of P1. We note that any edge from P1 to $\text{none}^{\text{PASSENGER}}$ will also be deleted. We note also that, by the semantics of this update program, a passenger can have any number of dependents.

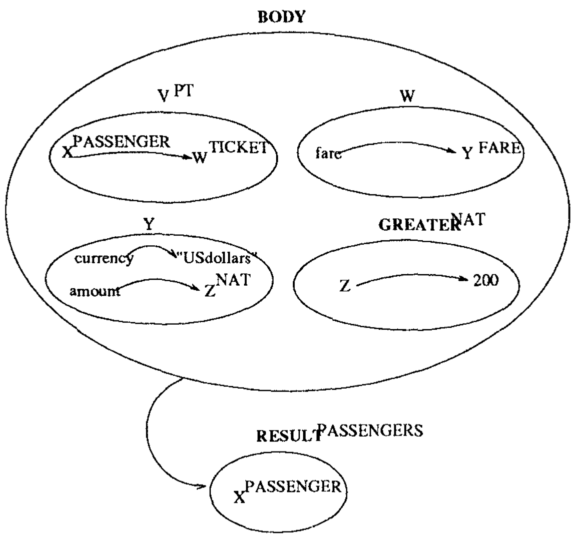


Fig. 20. Program to find passengers paying more than $200.

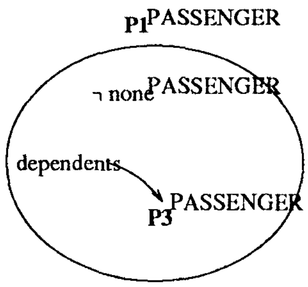


Fig. 21. Program to add P3 to the dependents of P1.

The program in Figure 22 replaces the old time of departure of the flight $\mathrm{FL1}^{\mathrm{FLIGHT}}$ by the new time $\mathrm{T1}^{\mathrm{TIME}}$ (by the type correctness of FL1, there must be some old time in FL1). This program illustrates how the cardinality of the time_of_dep attribute can be limited to 1.

Our final program in Figure 23 restructures the information about passengers so that it is stored in a number of mappings (c.f., functional data modeling [Shipman 1981]) rather than in one hypernode per passenger (c.f., relational data modeling).

We conclude this section by noting that it is not possible to write a Hyperlog rule which deletes an equation. Thus, in practice, garbage collection has to be achieved outside Hyperlog.

3.4 Operational Semantics of Hyperlog Programs

In this section we specify the operational semantics of Hyperlog via a 2-ary operator INFER(P, HR) where P is a Hyperlog program and where HR is a hypernode repository. INFER(P, HR) returns a new hypernode repository

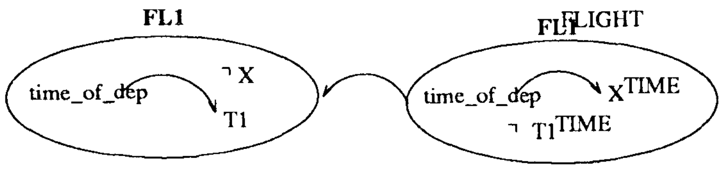


Fig. 22. Program to replace time of departure of flight FL1 by T1.

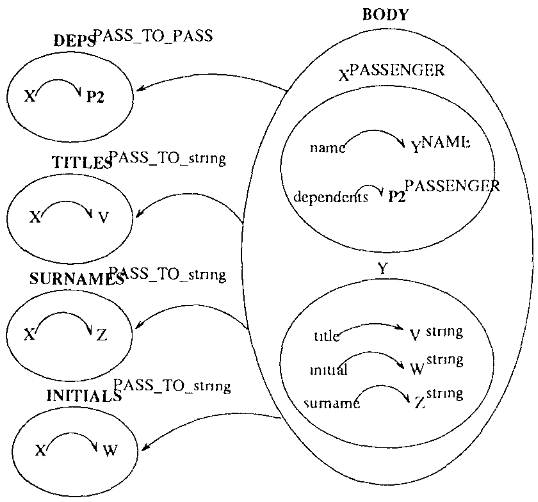


Fig. 23. Program to restructure passenger information into a number of mappings.

which differs from HR by the insertions and deletions which are inferred from HR by firing in parallel all the rules in P. A further operator, FIX(P, HR), computes the fixpoint of P with respect to HR by successive application of INFER(P, HR).

We begin by defining two binary operators on sets of equations, $\oplus$ and $\ominus$. Given two sets of equations HR and HR′, HR $\oplus$ HR′ consists of

(i) every equation $G = (N, E)$ in HR or HR′ such that $G \notin$ LABELS(HR) $\cap$ LABELS(HR′), and

(ii) for every pair of equations $G = (N, E)$ in HR and $G = (N', E')$ in HR′ with the same left-hand side, G, the equation $G = (N \cup N', E \cup E')$;

and HR $\ominus$ HR′ consists of

(i) every equation $G = (N, E)$ in HR such that $G \notin$ LABELS(HR′), and

(ii) for every pair of equations $G = (N, E)$ in HR and $G = (N', E')$ in HR′ with the same left-hand side, G, the equation $G = (N - N', E - (E' \cup \{(n_1, n_2) \in E | n_1 \in N' \text{ or } n_2 \in N'\}))$.

Now, let R be a rule $q_0 \leftarrow q_1, q_2, \ldots, q_n$ in a program P, where the head, q_0, is the query $Q = (N+ \cup N-, E+ \cup E-)$. Let θ be a match for the set of queries $\{q_1, q_2, \ldots, q_n\}$. Given θ, let θ^{new} be a substitution for NEW_R defined as follows:

(i) If $\text{NEW}_R = \emptyset$, $\theta^{\text{new}} = \emptyset$.
(ii) If there are one or more matches for $q_0\theta$, let θ^{new} be an arbitrary one of these matches.
(iii) Otherwise $\theta^{\text{new}} = \{Y_1/G_1, Y_2/G_2, \ldots, Y_k/G_k\}$ where $\text{NEW}_R = \{Y_1, Y_2, \ldots, Y_k\}$ and $G_1, \ldots, G_k$ are new labels superscripted with the appropriate type.

We denote the singleton set $\{Q\theta\theta^{\text{new}} = (N+, E+)\theta\}$ by $\text{POS}_R(\theta)$ and the singleton set $\{Q\theta\theta^{\text{new}} = (N-, E-)\theta\}$ by $\text{NEG}_R(\theta)$. We note that the equation in a $\text{NEG}_R(\theta)$ may have a right-hand side which is not a graph since it may be the case that there is an edge (n_1, n_2) in $E-$ in which case $n_1 \notin N-$ and $n_2 \notin N-$.

Finally, we define our main operator INFER(P, HR) to be

$$\text{HR} \oplus (\cup_{R \in P} \cup_\theta \text{POS}_R(\theta)) \ominus (\cup_{R \in P} \cup_\theta \text{NEG}_R(\theta))$$

provided the set of inferred insertions and the set of inferred deletions do not intersect, i.e., provided

$$(\cup_{R \in P} \cup_\theta \text{POS}_R(\theta)) \ominus (\cup_{R \in P} \cup_\theta \text{NEG}_R(\theta)) = \cup_{R \in P} \cup_\theta \text{POS}_R(\theta)$$

We do not wish any inferences to be made in the presence of such conflicts since we want a declarative semantics. So if the above equality does not hold we define INFER(P, HR) to be the old hypernode repository, HR.

We conclude by defining the operator FIX(P, HR) which computes the fixpoint of a hypernode program P with respect to a hypernode repository HR:

(i) $\text{FIX}^0(P, \text{HR}) = \text{HR}$;
(ii) $\text{FIX}^{i+1}(P, \text{HR}) = \text{INFER}(P, (\text{FIX}^i(P, \text{HR}))$;
(iii) $\text{FIX}(P, \text{HR}) = \text{FIX}^k(P, \text{HR})$ where
   (a) $\text{FIX}^{k+1}(P, \text{HR}) = \text{FIX}^k(P, \text{HR})$, and
   (b) $\forall j < k\ \text{FIX}^j(P, \text{HR}) \neq \text{FIX}^k(P, \text{HR})$.

The following proposition states that FIX(P, HR) is indeed a hypernode repository, i.e., it satisfies conditions H1 and H2 of Section 2.1. The proof of the proposition follows easily from the definition of INFER(P, HR), $\oplus$ and $\ominus$.

PROPOSITION 1. *FIX(P, HR) is a hypernode repository.*

Of course the computation of the fixpoint might not terminate. For example, the following program generates the successor Y^{NAT} of each natural number X^{NAT}, assuming the representation of the natural numbers we have used above and assuming that $0^{\text{NAT}} = (\{\text{none}^{\text{NAT}}\}, \emptyset)$ is already in the repository. Clearly, this program will carry on generating successors ad infinitum.

$$Y^{\text{NAT}} = (\{X^{\text{NAT}}\}, \varnothing) \leftarrow X^{\text{NAT}} = (\{Z^{\text{NAT}}\}, \varnothing)$$

The above rule contains a variable in its head that is not in its body, but this is not the only way in which nontermination can arise. For example, the following program inserts and deletes PER1 into C ad infinitum:

$$C^{\text{COUPLE}} = (\{\neg \text{PER1}, \text{none}^{\text{PERSON}}\}, \varnothing) \quad \leftarrow C = (\{\text{PER1}^{\text{PERSON}}\}, \varnothing)$$
$$C^{\text{COUPLE}} = (\{\text{PER1}, \neg \text{none}^{\text{PERSON}}\}, \varnothing) \quad \leftarrow C = (\{\neg \text{PER1}^{\text{PERSON}}\}, \varnothing)$$

The following proposition states that, if the computation of FIX(P, HR) does terminate, the resulting repository is unique up to the generation of new labels and the choice of the substitutions θ^{new}. The proof follows from the observation that these are the only nondeterministic steps in INFER(P, HR).

PROPOSITION 2. *FIX(P, HR) is unique, up to the drawing of new labels from* **L** *and the choice of* θ^{new}.

We conclude this section by noting that FIX(P, HR) as defined above ignores the type correctness of the new repository. In fact, a *static* type checking of programs can be performed before the fixpoint is computed: a program P is type correct if each rule $R \in P$ is type correct; a rule $q_0 \leftarrow q_1, q_2, \ldots, q_n$ is type correct if each query q_i is partially typed; finally, a query $Q^T = (N+ \cup N-, E+ \cup E-)$ is *partially typed* if the graph $(N+ \cup N-, E+ \cup E-)$ satisfies conditions T1 and T2 of Section 2.2 with respect to the type T.

We can make a number of observations. First, verifying that the queries in the body of each rule are partially typed prevents the evaluation of programs which are a priori type incorrect. Second, if the queries in the heads of rule are partially typed and contain only insertions, the hypernode repository FIX(P, HR) must be well typed (we recall from Section 3.3 that the null graph null(T) is assumed for dangling labels of type T). Third, deletions in rule heads may cause conditions (T3) and (T4) of Section 2.2 to be violated. This situation should not be allowed to occur, either by signaling a run-time error or by inserting (as part of the $\ominus$ operator) into partially typed graphs the appropriate nodes and edges from null(T). In order to simplify program specification, we have adopted the latter solution.

3.5 Efficiency of Inference

In this section we examine the efficiency of the INFER(P, HR) operator. We begin by observing that INFER(P, HR) is decidable since, given a repository HR, there are only a finite number of matches θ for a query q with respect to HR (due to the fact that there are only a finite number of constants that can be drawn from HR). Furthermore, for each rule $R \in P$ and each match θ for the body of R, θ^{new} is either chosen arbitrarily from a finite number of existing substitutions or obtained from a finite number of new labels. We now consider two aspects of the efficiency of INFER(P, HR): the complexity of finding a match for a query with respect to a hypernode repository and the potential number of matches for a query.

The next theorem states that finding a match for a query with respect to a hypernode repository is in general NP-complete.

THEOREM 2. *Finding a substitution, θ, which is a match for a query, q, with respect to a repository, HR, is NP-complete.*

PROOF. We first show NP-hardness by showing that this problem contains subgraph isomorphism, which is known to be NP-complete [Garey and Johnson 1979], as a subproblem. Let Q be a positive query of the form $G = (N+, E+)$, where the elements of $N+$ are all variables of the same type, and let there by an equation $G = (N', E')$ in HR. The result follows since θ is the required one-to-one mapping from $(N+, E+)$ to (N', E').

We next show that the problem is in NP. Given a query $Q = (N, E)$, we first guess a substitution θ for the query with respect to HR. If there is no equation in HR with left-hand side $Q\theta$ we are done. It remains to show that testing whether the equation $Q\theta = (N', E') \in$ HR is a match for the query $Q = (N, E)$ can be performed in a time polynomial in the size of (N, E) and (N', E'). The result follows since, on examining the definition of a match given in Section 3.2, we see that the testing can be performed in a time proportional to $|N||N'| + |E||E'|$. □

Despite this negative result, finding a match is less expensive in the case of certain graphs. For example, in the case that the graphs in the repository and the graphs in queries are *trees*, the problem can be solved in polynomial time in the size of the repository [Garey and Johnson 1979]. In practice, much data is record based, and so the corresponding graphs in the repository are forests (see, for example, the graphs in previous examples). Each such forest is equivalent to one tree whose root is the label of the graph, and so matching queries is tractable.

With respect to the number of matches for a query, there may exist an exponential number of matches: for example, given a query $G = (N+, \emptyset)$ such that the elements of $N+$ are variables of the same type and an equation $G = (N, \emptyset)$, the number of matches is $|N|!/(|N| - |N+|)!$. Negated nodes in queries can also lead to complexity. Consider for example matching the following rule body:

$$\leftarrow \mathrm{PER1}^{\mathrm{PERSON}} = (\{\neg Y^{\mathrm{string}}\}, \emptyset)$$

Clearly, there may be a large number of matches for Y (all the string constants in the database which are not the name of person PER1). This problem can be avoided by not allowing variables to appear negatively in the body of a rule without also appearing positively there. Then, given a rule $q_0 \leftarrow q_1, q_2, \ldots, q_n$, if any variable appearing in some N_i- also appears in some N_j+, we can construct substitutions θ for $\{q_1, q_2, \ldots, q_n\}$ by matching all positive information first (this technique is commonly known as *range restriction* [Abiteboul and Kanellakis 1989]). For example, we can range restrict the strings in the above rule body to be names of people thus:

$$\leftarrow \mathrm{PER1}^{\mathrm{PERSON}} = (\{\neg Y^{\mathrm{string}}\}, \emptyset), X^{\mathrm{PERSON}} = (\{\mathrm{name}, Y\}, \{\mathrm{name} \rightarrow Y\})$$

Negative variables are not the only problem with negative information: negative constant nodes can also lead to additional complexity when they occur within a query with a variable label. Consider, for example, matching the following rule body:

$$\leftarrow X^{\mathrm{PERSON}} = (\{\neg \text{“Jim”}\}, \emptyset)$$

In such cases the Hyperlog evaluator can at least make use of the type information to search for matches only within hypernodes of type PERSON.

Reducing the cost of finding all matches for positive information is more problematic. Clearly, given a query $Q = (N+ \cup N-, E+ \cup E-)$ the more variables there are in $N+$, the greater the number of possible substitutions for these variables. Again, the type tags of the variables narrow down the number of choices. The edge information (if any) is also of help here. Also, for record-based data whose attribute values are polynomially bounded (e.g., single-valued attributes) the number of matches for a query is polynomial in the size of the repository.

3.6 Expressiveness of Computation

Clearly, Hyperlog is a powerful language with respect to its expressiveness of computations and updates. In fact, it is both computationally complete and update complete.

We first demonstrate the computational completeness of Hyperlog by showing that it can simulate *counter programs*, which are known to be computationally complete [Harel 1987]. Counter programs manipulate natural numbers which are stored in variables called *counters*. Four operations are allowed on counters: $X := 0$, $X := Y$, $X := X + 1$ and $X := X - 1$, where X and Y are counters and $:=$ denotes assignment. In addition, counter programs support sequential composition and a goto statement conditional on some counter variable being 0.

We can simulate counters in Hyperlog by equations with distinguished left-hand sides, $\mathrm{CTR}^{\mathrm{NAT}}$, $\mathrm{CTR1}^{\mathrm{NAT}}$, $\mathrm{CTR2}^{\mathrm{NAT}}, \ldots$ say. We recall that the natural numbers are represented as successive nestings of the primitive node $\mathrm{none}^{\mathrm{NAT}}$, where $0 = (\{\mathrm{none}^{\mathrm{NAT}}\}, \emptyset)$. We can sequence the firing of rules in a counter program by using a set of distinguished labels, $\mathrm{STEP1}^{\mathrm{STEP}}$, $\mathrm{STEP2}^{\mathrm{STEP}}, \ldots$. The current step is contained in the node set of a further hypernode with label $\mathrm{SEQ}^{\mathrm{STEPS}}$, where $\mathrm{STEPS} = (\{\mathrm{STEP}\}, \emptyset)$. At the start of each program, this hypernode is assumed to be $\mathrm{SEQ} = (\{\mathrm{none}^{\mathrm{STEP}}\}, \emptyset)$.

For example, assigning zero to the counter CTR is achieved by inserting 0 into its node set and deleting any nonzero element already there:

$$\mathrm{CTR} = (\{\neg X, 0\}, \emptyset) \leftarrow \mathrm{CTR} = (\{X^{\mathrm{NAT}}\}, \emptyset), X = (\{\neg \mathrm{none}^{\mathrm{NAT}}\}, \emptyset)$$

Assigning the value of CTR to a counter CTR1 is achieved by the following rule:

$$\mathrm{CTR1} = (\{\neg X, Y\}, \emptyset) \leftarrow \mathrm{CTR1} = (\{X^{\mathrm{NAT}}\}, \emptyset), \mathrm{CTR} = (\{Y^{\mathrm{NAT}}\}, \emptyset)$$

Adding one to CTR is achieved by the following rules, which may generate a new natural number, Y^{NAT}:

$$\begin{aligned} Y^{\text{NAT}} &= (\{X\}, \emptyset) \leftarrow && \text{CTR} = (\{X^{\text{NAT}}\}, \emptyset) \\ \text{CTR} &= (\{Y, \neg X\}, \emptyset) \leftarrow && \text{CTR} = (\{X^{\text{NAT}}\}, \emptyset), Y^{\text{NAT}} = (\{X\}, \emptyset) \end{aligned}$$

Subtracting one from CTR is achieved by the following rule:

$$\text{CTR} = (\{X, \neg Y\}, \emptyset) \leftarrow \text{CTR} = (\{Y^{\text{NAT}}\}, \emptyset), Y = (\{X^{\text{NAT}}\}, \emptyset)$$

Testing CTR for zero is achieved by using the query CTR = $(\{0\}, \emptyset)$ in the body of a rule. Finally, sequential firing of rules and conditional goto are achieved by associating STEPs with rules and by updating the SEQ hypernode with the current STEP. For example, assuming a program with four steps

STEP1: CTR := 0
STEP2: goto STEP4 if CTR = 0
STEP3: ...
STEP4: ...

the goto statement at STEP2 is simulated by the following rule:

$$\text{SEQ} = (\{\text{STEP4}, \neg \text{STEP2}\}, \emptyset) \leftarrow \text{CTR} = (\{0\}, \emptyset), \text{SEQ} = (\{\text{STEP2}\}, \emptyset)$$

We conclude this section by examining the expressiveness of Hyperlog with respect to database updates. We first define what an update is in our context and then define the concept of update completeness, by analogy to previous work in relation databases [Abiteboul and Vianu 1988; Chandra and Harel 1980].

Given a type repository TR, we define the set *inst*(TR) to contain all hypernode repositories which are well typed with respect to TR. We define an *update* to be a partial recursive mapping from *inst*(TR) to *inst*(TR) that is *C-generic*. C-genericity was discussed in Hull and Su [1989] and intuitively means that, apart from a set of distinguished constants C (which may be the empty set), only the *structure* of a database is relevant to an update, not the *values* of the constants in the database. In our case an update, U, is C-generic if the following holds: given a finite set C of constants whose types are contained in PRIM(TR) $\cup$ LABELS(TR), for each HR $\in$ *inst*(TR) and each isomorphism ρ that maps primitive nodes to primitive nodes, labels to labels, and is invariant on C, $\rho(U(\text{HR}))$ is equal to $U(\rho(\text{HR}))$ up to a renaming of newly generated labels. The set C may be thought of as the constants (primitive nodes or labels) which appear explicitly in the update program.

Thus, a query language is *update complete* for the Hypernode Model if it precisely defines the set of updates as defined above. The update completeness of Hyperlog in particular follows from similar results in Abiteboul and Kanellakis [1989], Abiteboul and Vianu [1988], Chandra and Harel [1980], and Hull and Su [1989] for logic-based languages of comparable semantics.

3.7 Using Hyperlog for Database Browsing

Up to now we have considered querying (and updating) the database by partially specifying the *contents* of hypernodes. In contrast, browsing allows the user to navigate through the *structure* of the database independent of actual values. In the case of the Hypernode Model, navigation can follow edges either forward or backward; it can descend into a node from a parent graph; or it can ascend into a parent graph from a node. We show below how these navigational operators can be implemented in Hyperlog. In general, it will be difficult for the user to predict the types of the hypernodes that will be encountered while browsing to the database. So in order to facilitate browsing we introduce the type ANY as a supertype of every type, i.e., we consider any hypernode or primitive node to be of type ANY.

We first define three types:

$$\begin{aligned} \mathrm{CONTEXT} &= (\{\mathrm{CURRENT\_HYP}, \mathrm{CURRENT\_NODE}\}, \emptyset) \\ \mathrm{CURRENT\_HYP} &= (\{\mathrm{ANY}\}, \emptyset) \\ \mathrm{CURRENT\_NODE} &= (\{\mathrm{ANY}\}, \emptyset) \end{aligned}$$

Instances of type CONTEXT will typically contain two nodes, one of type CURRENT_HYP which contains a hypernode and the other of type CURRENT_NODE which contains a specific node within this hypernode. The current context can thus be recorded in a hypernode

$$\begin{aligned} &\mathrm{CUR\_CONTEXT}^{\mathrm{CONTEXT}} \\ &\quad = (\{\mathrm{CUR\_HYP}^{\mathrm{CURRENT\_HYP}}, \mathrm{CUR\_NODE}^{\mathrm{CURRENT\_NODE}}\}, \emptyset) \end{aligned}$$

where CUR_HYP contains the current hypernode in the navigation and where CUR_NODE contains the specific node within the current hypernode currently being browsed.

The current hypernode can be updated from a hypernode OLD, say, to a hypernode NEW by the rule

$$\mathrm{CUR\_HYP} = (\{\mathrm{NEW}, \neg\,\mathrm{OLD}\}, \emptyset) \leftarrow$$

Similarly, the current node can be updated from OLD to NEW by the rule

$$\begin{aligned} &\mathrm{CUR\_NODE} = (\{\mathrm{NEW}, \neg\,\mathrm{OLD}\}, \emptyset) \leftarrow \\ &\qquad\qquad \mathrm{CUR\_HYP} = (\{X\}, \emptyset),\ X = (\{\mathrm{NEW}\}, \emptyset) \end{aligned}$$

We observe that this rule verifies the new current node is indeed in the node set of the current hypernode.

In order to navigate forward, we can store in a hypernode $\mathrm{CUR\_OUT}^{\mathrm{CURRENT\_NODE}}$ the nodes connected to the current node by edges outgoing from it: we initialize the previous contents of CUR_OUT using the rule

$$\mathrm{CUR\_OUT} = (\{\neg\, X, \mathrm{none}^{\mathrm{ANY}}\}, \emptyset) \leftarrow \mathrm{CUR\_OUT} = (\{X\}, \emptyset)$$

and we store the "next" nodes in CUR_OUT using the rule

$$\begin{aligned} &\mathrm{CUR\_OUT} = (\{Y, \neg\,\mathrm{none}^{\mathrm{ANY}}\}, \emptyset) \leftarrow \\ &\qquad\qquad \mathrm{CUR\_NODE} = (\{X\}, \emptyset), \mathrm{CUR\_HYP} \\ &\qquad\qquad = (\{Z\}, \emptyset), Z = (\{X, Y\}, \{X \to Y\}) \end{aligned}$$

Similarly, in order to navigate backward, we can store in a hypernode $\mathrm{CUR\_IN}^{\mathrm{CURRENT\_NODE}}$ the nodes connected to the current node by edges incoming to it: we initialize the previous contents of CUR_IN as for CUR_OUT above, and we store the "previous" nodes in CUR_IN using the rule

$$\begin{aligned} &\mathrm{CUR\_IN} = (\{Y, \neg\,\mathrm{node}^{\mathrm{ANY}}\}, \emptyset) \leftarrow \\ &\qquad\qquad \mathrm{CUR\_NODE} = (\{X\}, \emptyset), \mathrm{CUR\_HYP} \\ &\qquad\qquad = (\{Z\}, \emptyset), Z = (\{X, Y\}, \{Y \to X\}) \end{aligned}$$

In order to navigate upward, we store in a hypernode CUR_UP$^{\text{CURRENT_NODE}}$ all the hypernodes containing the current hypernode: we initialize the previous contents of CUR_UP and use the rule

$$\text{CUR\_UP} = (\{Y, \neg\text{none}^{\text{ANY}}\}, \emptyset) \leftarrow \text{CUR\_HYP} = (\{X\}, \emptyset), Y = (\{X\}, \emptyset)$$

Finally, in order to navigate downward, we store in a hypernode CUR_DOWN$^{\text{CURRENT_NODE}}$ all the nodes contained in the node set of the current node (if this is not a primitive node): we initialize the previous contents of CUR_DOWN and use the rule

$$\text{CUR\_DOWN} = (\{Y, \neg\text{none}^{\text{ANY}}\}, \emptyset) \leftarrow$$
$$\text{CUR\_NODE} = (\{X\}, \emptyset), X = (\{Y\}, \emptyset)$$

Browsing using Hyperlog was investigated further in Dearden [1990]. In particular, it was shown that Hyperlog can support the declarative querying of the content and structure of a Hypertext database. This database was constructed by associating hypernodes with fragments of text and by using further hypernodes to store named links between these fragments. A "history" hypernode records the user's navigation through the database. A number of alternative "trails" can be set up and stored. The navigational functions supported include display of a hypernode (and any associated text) and the four operators described above.

4. COMPARISON WITH RELATED WORK

In this section we compare the Hypernode Model and Hyperlog with related languages and models. We begin with the logic-based database language IQL [Abiteboul and Kanellakis 1989] from which the semantics of Hyperlog are partly derived. We next consider three recent graph-based data models [Consens and Mendelzon 1990; Gyssens et al 1990; Kuper and Vardi 1984]. Finally we consider recent work on hypergraph-based models [Levene and Poulovassilis 1991; Tompa 1989; Watters and Shepherd 1990].

IQL incorporates object identities into a typed rule-based query language which is update complete. The fixpoint semantics of Hyperlog are similar to those discussed in Abiteboul and Kanellakis [1989] and Abiteboul and Vianu [1989], but our label generation semantics differ from IQL's invention of object identities, in that we generate new labels as a necessary consequence of new graphs being inferred, whereas in IQL the generation of an object identity and the assignment of a value to it are independent events. Also, IQL's types are constructed using tuple, set, union, and intersection constructors while Hyperlog has one general-purpose graph constructor which can simulate all of these.

We next compare the Hypernode Model with three recent graph-based data models: the Logical Data Model (LDM) [Kuper and Vardi 1984], GOOD [Gyssens et al. 1990], and Graphlog [Consens and Mendelzon 1990]. In LDM only database schemas are directed graphs: instances consist of 2-column tables, each of which associates entities of a particular type with their values. Also, LDM's schema graphs use three types of node, basic (for primitive data types), composition (for tuple types), and collection (for set types), whereas we can represent tuple types and set types by our one general-purpose graph constructor. Graphlog is a query language operating on a database which comprises a directed labeled graph (a semantic net). The edges in this graph represent predicates. Unlike Hyperlog, Graphlog queries are formulated as graphs whose edges are annotated with predicates, transitive closures thereof, or, more generally, regular expressions. These query graphs are matched against the database graph and return subgraphs thereof. GOOD is a graphically represented, functional data model [Shipman 1981] with an associated transformation language. GOOD embeds semantics into the nodes and edges of this graph, nodes being printable or nonprintable and edges being single-valued or multivalued. The queries of GOOD's transformation language are graphs called *patterns*, which match subgraphs of the total instance graph, c.f., our matching of queries against a hypernode repository. In contrast to Hyperlog's rule-based updates, GOOD's instance graph is updated by five graphically represented primitive operations (add or delete a node or an edge and an operation called "abstraction") which can be incorporated into patterns.

In summary, a feature common to all these models is that the database consists of a single flat graph. This has the drawback that, in practice, complex objects consisting of many interconnected nodes are hard to present to the user in a clear way. In contrast, a hypernode database consists of a set of nested graphs. This unique feature of our model provides inherent support for data abstraction and the ability to represent each real-world object as a separate database entity. GOOD does allow for an "abstraction" operation, but this generates a nonprintable entity and connects it to other related entities at the same level, e.g., our nesting of a set of graphs within a graph. Unlike GOOD and Graphlog, we do not label edges in the Hypernode Model. However, we can attain the same data-modeling expressiveness by encapsulating edges which would have the same label in GOOD or Graphlog within one hypernode with a similar label. For example, we can represent the set of GOOD edges

$$P1 \rightarrow^{HAS\_TICKET} T1$$
$$P2 \rightarrow^{HAS\_TICKET} T2$$
$$P3 \rightarrow^{HAS\_TICKET} T3$$

by the hypernode

$$HAS\_TICKET = (\{P1, T1, P2, T2, P3, T3\}, \{P1 \rightarrow T1, P2 \rightarrow T2, P3 \rightarrow T3\})$$

We conclude this section with a review of recent work on hypergraph-based data models [Levene and Poulovassilis 1991; Tompa 1989; Watters and Shepherd 1990], and a comparison of this work with our model. We first observe that hypergraphs can be modeled by hypernodes by encapsulating the contents of each hyperedge within a further hypernode. In contrast, the multilevel nesting provided by hypernodes cannot easily be captured by hypergraphs.

In Tompa [1989], hypergraphs are used to model page-oriented Hypertext databases. The nodes of a hypergraph are associated with pages of information. Each hyperedge consists of a related set of labeled directed edges. Nodes and directed edges can be shared between hyperedges. Querying of a hypergraph is navigational and uses a number of predefined operators: browsing forward or backward along directed edges from a set of marked nodes, marking a new set of nodes, reading the set of pages associated with the current marked nodes, querying the current state, and saving and resetting current states. Views can be created, and the database hypergraph can be updated by a number of further primitive operators. Unlike the Hypernode Model, hypergraphs are not typed, and so updates are not semantically constrained. Also, unlike Hyperlog, querying by database content is not supported.

In Watters and Shepherd [1990], a hypergraph-based model of data access is presented which aims to integrate browsing and querying. In this model, entities are represented by nodes and relationships between them by hyperedges. The resulting hypergraph is transient, lasting for the duration of a query session. It starts off consisting of one hyperedge containing all the database entities, and further hyperedges are added to it in response to user queries. At any stage, the hypergraph can be traversed by moving within hyperedges and from hyperedge to hyperedge via a common node.

There are a number of differences between this work and our own. First, all the database entities are assumed to be of the same type and are stored as tuples in a single flat relation. Second, the attributes of entities are not represented graphically within the hypergraph and exist only in the underlying relation. Third, although browsing is graph based, querying is not—it consists of specifying boolean-valued expressions in the values of attributes—hence, a hybrid model of browsing and querying is obtained.

In Levene and Poulovassilis [1991] we described a data model called GROOVY (Graphically Represented Object-Oriented data model with Values). In GROOVY, real-world entities are represented by means of instances of object schemas. We showed that the representation of object schemas by means of hypergraphs leads to a natural formalization of the notions of subobject sharing and structural inheritance. We showed also how instances of object schemas can be represented by hypergraphs labeled with object identifiers. GROOVY is a conceptual data model which influenced the development of the Hypernode Model. It has been superseded by our more recent work on types, Hyperlog, and implementation.

5. SYSTEM ARCHITECTURE AND IMPLEMENTATION

We are currently coming to the end of a two-year project whose goal is to implement a prototype DBMS based on the Hypernode Model and to tailor it to the needs of Hypertext databases. The architecture of our system is shown in Figure 24. In this architecture, the *Storage Manager* stores hypernodes, types, and programs while the *Index Manager* supports efficiently three operations:

(i) given a label, G, return the unique graph (N, E) such that $G = (N, E)$,

(ii) given a primitive node, n, return the set of labels $\{G_1, \dots, G_r\}$ such that for each equation $G_i = (N_i, E_i)$, $n \in N_i$,

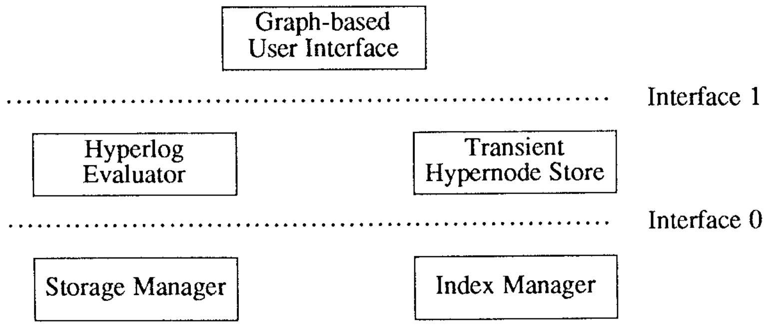


Fig. 24. The Hypernode Database System architecture.

(iii) given a label, G, return the set of labels $\{G_1, \ldots, G_r\}$ such that for each equation $G_i = (N_i, E_i)$, $G \in N_i$.

A detailed description of the Storage Manager appears in Tuv et al. [1992]. Briefly, the Storage Manager supports a number of object stores, each object store containing graphs of one type. Two object stores are reserved for the storage of types and programs. Associated with each object store is a label table which maps labels to the physical addresses of the graphs they define, thereby implementing operation (i) above. Operation (ii) is implemented using a simple prefix B-tree, and operation (iii) is implemented by an extendible hashing scheme.

These operations are invoked by the *Hyperlog Evaluator* during its matching of queries. The evaluator computes the fixpoint of a program with respect to the repository, after verifying that the program is correctly typed. Updated hypernodes are amassed in the *Transient Hypernode Store* during each inference step. The evaluator currently uses *bottom-up*, *naive* [Ullman 1988] evaluation of Hyperlog programs although we are now looking at optimizing the fixpoint computation by drawing on existing optimization techniques for logic database languages, such as *seminaive* evaluation.

6. SUMMARY

We have presented the Hypernode Model, a graph-based data model which stores nested graphs in the form of equations and manipulates them via a rule-based language. The key innovations of the model are:

— its formal foundation on graphs and set theory,
— its use of graphs throughout all levels, from the user interface down to the physical level,
— its inherent support for data-modeling concepts such as object identity, complex objects, and encapsulation,
— its provision for types and type checking,

—its associated query language Hyperlog which can support both querying and browsing and which allows both derivations and database updates,

—its uniform storage of data (hypernodes), metadata (types), and procedural data (Hyperlog programs).

We have examined the efficiency of type checking and have shown that it can be performed in polynomial time. We have also examined the expressiveness of representation, computation, and update of our model and have shown that Hyperlog is computationally and update complete. Although the evaluation of Hyperlog programs is intractable in the general case, we have discussed cases when evaluation becomes tractable. We have compared our model with other graph-based models. Our comparison has highlighted the advantages of nested graphs, both at the type and the instance levels. Finally, we have briefly discussed a prototype DBMS architecture and implementation. Our current research effort is directed toward tailoring our hypernode DBMS to the needs of Hypertext. This includes optimization of Hyperlog, support for versioning, and provision of special-purpose access methods to implement more efficiently the browsing and text retrieval operations.

REFERENCES

ABITEBOUL, S., AND KANELLAKIS, P. C. 1989. Object identity as a query language primitive. In *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (Portland, Ore., 1989). ACM, New York, 159–173.

ABITEBOUL, S., AND VIANU, V. 1988. Procedural and declarative database update languages. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Austin, Tex., 1988). ACM, New York, 240–250.

ABITEBOUL, S., AND VIANU, V. 1989. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proceedings of the Symposium of Logic in Computer Science*. 71–79.

ACZEL, P. 1988. *Non-Well-Founded Sets*. Lecture notes, no. 14. Center for the Study of Language and Information (CSLI), Stanford, Calif.

BEERI, C. 1989. Formal models for object oriented databases. In *Proceedings of the International Conference on Deductive and Object-Oriented Databases*. 370–395.

BERGE, C. 1973. *Graphs and Hypergraphs*. North-Holland, Amsterdam.

CERI, S., GOTTLOB, G., AND TANCA, L. 1990. *Logic Programming and Databases. Surveys in Computer Science*. Springer-Verlag, New York.

CHANDRA, A. K., AND HAREL, D. 1980. Computable queries for relational data bases. *J. Comput. Syst. Sci. 21*, 156–178.

CHEN, P. P-S. 1976. The Entity-Relationship Model—towards a unified view of data. *ACM Trans. Database Syst. 1*, 1, 9–36.

CODD, E. F. 1979. Extending the database relational model to capture more meaning. *ACM Trans. Database Syst. 4*, 1, 397–434.

CONKLIN, J. 1987. Hypertext: An introducton and survey. *IEEE Comput. 20*, 9, 17–41.

CONSENS, M. P., AND MENDELZON, A. O. 1990. Graphlog: A visual formalism for real life recursion. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Nashville, Tenn., 1990). ACM, New York, 404–416.

DEARDEN, A. 1990. A hypertext database implemented using the Hypernode Model. Masters thesis, Dept. of Computer Science, Univ. College London, London.

GAREY, R. G., AND JOHNSON, D. S. 1979. *Computers and Intractability, A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York.

GRIFFITH, R. L. 1982. Three principles of representation for semantic networks. *ACM Trans. Database Syst. 7*, 3, 417–442.

GYSSENS, M., PAREDAENS, J., AND VAN GUCHT, D. V. 1990. A graph-oriented object database model. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Nashville, Tenn., 1990). ACM, New York, 417–424.

HAREL, D. 1988. On visual formalisms. *Commun. ACM 31*, 5, 514–530.

HAREL, D. 1987. *Algorithmics—The Spirit of Computing*. Addison-Wesley, Reading, Mass.

HULL, R., AND KING, R. 1987. Semantic database modeling: Survey, applications, and research issues. *ACM Comput. Surv. 19*, 3, 201–260.

HULL, R., AND SU, J. 1989. Untyped sets, invention, and computable queries. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Philadelphia, Penn., 1989). ACM, New York, 347–359.

KIM, W. 1990. Object-oriented databases: Definition and research directions. *IEEE Trans. Knowl. Data Eng. 2*, 3, 327–341.

KUPER, G. M., AND VARDI, M. Y. 1984. A new approach to database logic. In *Proceedings of the ACM Symposium on Principles of Database Systems* (Waterloo, 1984). ACM, New York, 86–96.

LEVENE, M., AND POULOVASSILIS, A. 1991. An object-oriented data model formalized through hypergraphs. *Data Knowl. Eng. 6*, 3, 205–224.

LEVENE, M., AND POULOVASSILIS, A. 1990. The hypernode model and its associated query language. In *Proceedings of the 5th JCIT*. IEEE Computer Society Press, Los Alamitos, Calif., 520–530.

NAQVI, S., AND TSUR, S. 1989. *A Logical Language for Data and Knowledge Bases*. Computer Science Press, New York.

PARENT, C., AND SPACCAPIETRA, S. 1989. Complex object modelling: An entity-relationship approach. In *Nested Relations and Complex Objects in Databases*, S. Abiteboul, P. C. Fischer, and H.-J. Schek, Eds., Springer-Verlag, Berlin, 272–296.

PRZYMUSINSKA, H., AND PRZYMUSINSKI, T. 1990. Semantic issues in deductive databases and logic programs. In *Formal Techniques in Artificial Intelligence, a Sourcebook*, R. B. Banerji, Ed. Elsevier Science, Amsterdam, 321–367.

REITER, R. 1978. On closed world databases. In *Logic and Databases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 55–76.

SCHEK, H.-J., AND SCHOLL, M. H. 1986. The relational model with relation-valued attributes. *Inf. Syst. 11*, 2, 137–147.

SHIPMAN, D. 1981. The functional data model and the data language DAPLEX. *ACM Trans. Database Syst. 6*, 1, 140–173.

SHRIVER, B., AND WEGNER, P., Eds. 1987. *Research Directions in Object-Oriented Programming*. MIT Press, Cambridge, Mass.

TOMPA, F. W. 1989. A data model for flexible hypertext database systems. *ACM Trans. Inf. Syst. 7*, 1, 85–100.

TUV, E., POULOVASSILIS, A., AND LEVENE, M. 1992. A storage manager for the Hypernode Model. In *Proceedings of BNCOD-10—Advances in Database Systems*. Lecture Notes in Computer Science, vol. 618. Springer-Verlag, New York, 59–77.

ULLMAN, J. D. 1988. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, Rockville, Md.

WATTERS, C., AND SHEPHERD, M. A. 1990. A transient hypergraph-based model for data access. *ACM. Trans. Inf. Syst. 8*, 2, 77–102.

WONG, K., AND LOCHOVSKY, F., Eds. 1989. *Object-Oriented Languages, Applications, and Databases*. ACM Press Frontier Series, New York.

Received July 1991; accepted November 1992