Database Query Navigation*

B. Amann[1] and M. Scholl[2,1]

[1] INRIA F-78153 Le Chesnay Cedex France
[2] Cedric/CNAM 292 rue St Martin 75141 Paris Cedex 03 France
Email: <Bernd.Amann|Michel.Scholl>@inria.fr

Abstract. In [AS92] a database model was defined which uses regular expressions over data types to formulate queries. We discuss its expressive power and utilize it as an underlying data model for designing general database navigation mechanisms. The application we have in mind is hypertext navigation and querying. However we believe that these mechanisms are also applicable on top of other data models such as the relational or object-oriented database models.

1 Introduction

Graphs are a natural way of describing the structure of information in data intensive applications. The CODASYL DBTG network model [COD71], entity-relationship models [Che76], and semantic data models [AH87, HK87] are examples of paradigms which make an intensive use of graph based diagrams, useful not only for representing database schemas, but also as a help for navigating through data. Although relational query languages such as SQL are declarative, a complex query including a large number of relations, say more than 3, is more naturally conceptualized and prepared by navigating among relations, and a graph representation of the schema becomes extremely useful. This is even more obvious with object-oriented databases: with query languages such as O_2SQL [BCD89] or XSQL [KKS92] the user navigates through classes (connected by composition links) by means of *path expressions*. GraphLog [CM90, CM89, CMW88] and GOOD [GPG90] are also based on a graph representation of data, but in contrast with other query languages, queries are graph patterns in which nodes and edges are matched against subgraphs of the database. [BK90, BRS90, Güt91] are further recent works in which data is manipulated as nodes and edges of a graph.

Hypertext systems is another example of graph structured information systems based on navigation. (Multimedia) data is stored in nodes or *documents* which are connected by *links*. More precisely, a link connects spans of characters, called *anchors* or *buttons*, included in documents. Navigation through links is the only way of accessing data in such systems. In contrast with database systems, where data is strongly typed, hypertext documents are weakly typed, and the user not only navigates through documents independently of their type, but also can add a new document on-line without necessarily deciding whether it pertains to a given group of documents (is the instance of a given type). Navigation can be supported by a graph browser displaying the current user position in the hypertext graph representation (map). Clearly, such maps become tangled when the number of nodes and edges passes some limit (a few dozens of nodes [Con87]). This is why it is necessary to provide documents, and possibly links, with strong typing [NN91], and use query languages for restricting the search space in navigational graphs [Hal88]. Querying, then, allows us to define a variety of navigation strategies. The objective of this paper is to investigate the mechanisms which are necessary for implementing such strategies.

In [AS92] we have defined a data model and query language, called Gram and based on regular (path) expressions over data types. We shall use Gram as the underlying data model for the presentation of our navigation mechanisms, but these should also be applicable on top of other data models such as, for instance, the model presented in [BK90], GraphLog [CM90] or any relational or object-oriented data model.

We first introduce Gram through examples of data and queries in Section 2. A formal presentation of the model and the algebra is given in Section 3. Gram is based on an algebraic language and supports a restricted form of recursion. In Section 4 we compare its expressive power to that of first-order logic plus transitive closure (FO+TC). Section 5 is devoted to the application of query languages to user navigation in graph based databases. We introduce a few basic mechanisms, namely user state, virtual link or short-cut, reverse link and user history. Section 6 introduces a better information structuring through the definition of a zooming function for connecting several databases: zooming allows us in particular reutilization of existing databases. It is also useful for modeling hierarchical relationships.

* Work partially supported by the BRA Esprit Project *Amusing.*

2 Gram by Examples

A Gram database schema is represented by a labeled graph. The node labels correspond to document types and the edges are labeled by link types. Figure 1 shows a schema of a travel agency hypertext. A document can have one of the types Journey, Stop, City, Hotel, Monument and Guide. Links are also typed and may contain information such as a date (type date), an address (type addr), a flight number (type flight), or an airport (type in). Journeys are described in documents of type Journey which are linked to Stop documents, one for each city to be visited. From each stop we can reach city, hotel and monument information by following links of type in and addr. Hotels with a nice view on a monument are connected to the corresponding Monument document by a link of type view, which contains the number of free rooms having a view on the monument. For example a link *2* between document *Hotel_Montmartre* and monument *Sacre_Cœur* indicates that the Montmartre hotel has two free rooms with a nice view on Sacre Cœur. Documents of type Guide contain the names, photos and phone numbers of the tourist guides whose working plans are described by links of type date. For example, the fact that Bob is guiding the Orsay museum on August 3, 1993, is represented by a link *8/3/93* of type date connecting the document *Bob* to the Monument document *Orsay_Museum*.

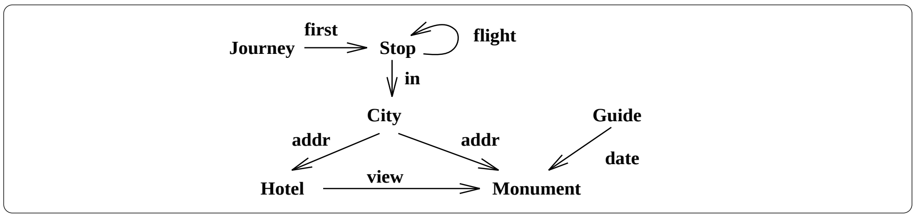

Figure 1: A Travel Agency Schema

The graph query language is based on a path[3] algebra and makes it possible to

1. select walks according to regular expressions over node and edge types.
2. select walks according to node and edge values.
3. construct new walks by projection and concatenation.
4. join walks sharing at least one node.

For illustration, assume we have a relational database with two relations `ADDR(city,addr,monument)` and `VIEW(hotel,view,monument)`. The following SQL query gets all hotels with a view on monuments on the Champ de Mars in Paris:

```
SELECT hotel
  FROM ADDR, VIEW
 WHERE addr="champ_de_mars" AND
       ADDR.monument=VIEW.monument
```

Each tuple in relation `ADDR` (relation `VIEW`) can also be seen as a walk *satisfying* the regular expression (r.e.) City addr Monument (r.e. Hotel view Monument). An example corresponding to the tuples `[Paris,champ_de_mars, Tour_Eiffel]` and `[Hotel_Trocadero,2,Tour_Eiffel]` is depicted in Figure 2. The join between both tuples is represented by the shared node `Tour_Eiffel`. In the Gram model this set of walks is called a *hyperwalk*. As for walks, we will say that the hyperwalk in Figure 2 *satisfies the hyperwalk (regular) expression* City addr Monument+Hotel view Monument.

Figure 2: A Hyperwalk

[3] In the following we will use the term *walk* to denote paths in a Gram database.

With the Gram algebra to be defined in Section 3, the following query gets all monuments that can be seen from the Pullman hotel in Paris (a syntax a la SQL is used for clarity):

```
SELECT Monument
  FROM City addr Hotel view Monument
 WHERE City.name="Paris" AND
       Hotel.name="Pullman"
```

Walks satisfying the r.e. City addr Hotel view Monument are selected and restricted to the `WHERE` clause condition. Finally, we project these walks on their Monument node (document). The first query (hotels with a view on the Champ de Mars) is formulated in Gram as a selection/projection:

```
SELECT Hotel
  FROM City addr Monument + Hotel view Monument
 WHERE addr.street="champ_de_mars"
```

Note that the walks in the range of the query (`FROM` clause by City addr Monument + Hotel view Monument) have to share a node of type Monument. The schema in Figure 1 is cyclic (edge type flight). This allows the application of Kleene closure, as shown in the next example. The following query gets the journeys with at least one stop in Los Angeles:

```
SELECT Journey
  FROM Journey first(Stop next)*Stop in City
 WHERE City.name="Los_Angeles"
```

3 The Gram Data Model and Query Language

We briefly describe the Gram data model and its query language based on algebraic operations. For a complete presentation see [AS92, Ama94].

Database Schema: A *(graph database) schema* is a directed weakly connected[4] labeled multigraph $\mathcal{S} = (N_S, E_S, \psi_S, \lambda_S, T)$ where (1) N_S is a set of nodes and E_S is a set of edges, (2) ψ_S is an incidence function from E_S into $N_S \times N_S$ and (3) $T = T_N \cup T_E$ is a set of labels with a labeling function λ_S from $N_S \cup E_S$ into T. Additionally, S has to satisfy the *distinct node label* and *distinct edge label* properties: different nodes must have different labels and two nodes cannot be connected by two links with the same label.

We assume that each symbol t in T, called a type, is associated with a *domain* of values, denoted $dom(t)$. A set of values V is defined as the union of the domains of all types in T: $V = \bigcup_{t \in T} dom(t)$. We do not make any other assumptions on the type system T. For example, Figure 1 shows a schema with 6 nodes and 7 edges labeled by the types $T = \{$Journey, Stop, City, Hotel, Monument, Guide, first, flight, in, addr, date, view$\}$.

Graph Database: A *(graph) database* with schema $\mathcal{S}$ is a directed labeled multigraph $G = (N, E, \psi, \lambda, V)$ where (1) N is a set of nodes and E is a set of edges, (2) ψ is an incidence function from E into $N \times N$ and (3) V is a set of labels with a labeling function λ from $N \cup E$ into V. There exists a function τ from G to $\mathcal{S}$ associating with each node ν (each edge ϵ) in G a node $\tau(\nu)$ (an edge $\tau(\epsilon)$) in $\mathcal{S}$ such that the label of node ν (edge ϵ) is in the domain of the label of $\tau(\nu)$ ($\tau(\epsilon)$). Additionally, if ϵ is going from ν to ν' then $\tau(\epsilon)$ must go from $\tau(\nu)$ to $\tau(\nu')$. Note that a database G is not necessarily connected and there may be nodes without edge or more than one incoming or outgoing edge.

Walks and Hyperwalks: A *walk* in a (database) graph is an alternating sequence of nodes and edges $n_0 e_0 n_1 \ldots n_{i-1} e_{i-1} n_i$ beginning and ending with nodes. As a matter of fact, a walk the user traverses, might be related to information in other walks. Walks sharing nodes are combined into sets of walks, called *hyperwalks*.

[4] A directed graph is weakly connected if the corresponding undirected graph is connected [Har71]. From now on, connected will mean weakly connected.

The hyperwalk h ={*Paris.champ_de_mars.Tour_Eiffel, Hotel_Trocadero.2.Tour_Eiffel*} consists of two walks, starting in different nodes and ending in the same Monument node (Figure 2).

Walk Expression: A *walk expression* is a regular expression (r.e.) over T without alternation (+), whose language contains only alternating sequences of node and edge types, starting and ending with a node type.

Hyperwalk Expression: A regular expression over T is a *hyperwalk expression (hwe)* iff it can be rewritten as a sum of walk expressions, $r = \sum_{1 \leq i \leq n} r_i$ such that (1) the schema graph corresponding to r is *connected* and (2) if r_i and r_j share a node type t, then both in r_i and in r_j, t has only one occurrence (outside Kleene closure)[5].

Example 1. The r.e. City addr Hotel + City addr Monument is a hyperwalk expression, whereas Stop flight Stop + Hotel view Monument is not since the corresponding schema graph is disconnected (both walk expressions do not share any node type). The r.e. r =Stop flight Stop + Stop in City is not a hyperwalk expression since the shared node type occurs twice in the first walk expression (Condition 2). On the other hand, r' =Stop flight Stop' + Stop in City is a hyperwalk expression (the second occurrence of Stop has been renamed). Finally, note that the r.e. Stop(flight Stop)* + Stop in City is a hyperwalk expression: the second occurrence of Stop in the first term appears inside the Kleene closure of r.e. flight Stop.

Hyperwalk satisfying a Hyperwalk Expression: The *label* $\lambda(w)$ of a walk w is obtained by substituting each node and edge in the walk with its label. The *label* $\lambda(h)$ of a hyperwalk h is obtained by replacing each walk in h by its label. We define the following language over T:

$$\begin{aligned}
\mathcal{L}(t) &= dom(t) \text{ for all } t \in T. \\
\mathcal{L}(A.B) &= \mathcal{L}(A) \times \mathcal{L}(B) \text{ for all r.e. } A \text{ and } B. \\
\mathcal{L}(A+B) &= \mathcal{L}(A) \cup \mathcal{L}(B) \text{ for all r.e. } A \text{ and } B. \\
\mathcal{L}(A^*) &= \cup_{i \geq 0} \mathcal{L}(A^i) \text{ where } \mathcal{L}(A^0) = \{\epsilon\} \text{ for all regular expressions } A.
\end{aligned}$$

From now on, $\mathcal{L}(r)$ will be called the language of r. Let r be a walk expression and w be a walk whose label is in $\mathcal{L}(r)$: $\lambda(w) \in \mathcal{L}(r)$. With each occurrence of a type t in r we can associate some nodes or edges in w, called instances of t in w. For example, *Vienna* is an instance of City in the walk *Vienna.ring.Marriott* whose label is in $\mathcal{L}$(City addr Hotel).

Satisfaction: A hyperwalk $h = \{w_1, w_2, \ldots w_n\}$ *satisfies* a *hwe* $r = \sum_{1 \leq i \leq n} r_i$, denoted $h \models r$, iff (1) $\lambda(h) \subseteq \mathcal{L}(r)$ and (2) the walks in h form a connected graph. More precisely, for the occurrence of each node type t shared by r_i and r_j, w_i and w_j have to share the instance of type t.

Example 2. The set of walks {*Paris.champ_de_mars.Tour_Eiffel*, *Lydie.8/10/93.Tour_Eiffel*} is a hyperwalk satisfying hwe City addr Monument + Guide date Monument: both walks share *Tour_Eiffel*, which is the instance of node type Monument.

Instance of a Hyperwalk-Expression: The *instance* of a *hwe* r in a database G is defined as the set of hyperwalks in G satisfying r: $I(r) = \{h \mid h \in G \wedge h \models r\}$.

3.1 Hyperwalk Algebra

A *query* is an algebraic expression on sets of hyperwalks with as unary operations *projection, selection* and *renaming*, and as binary operations *join, concatenation*, as well as the usual set operations *union, intersection*, and *difference*.

Renaming: Hyperwalk expressions may have several occurrences of the same type. *Renaming* allows one to distinguish between the different occurrences of a given type in a given *hwe*. For example, h_1 ={*vie2.AF345.par2*} and h_2 ={*mun6.LH234.par6*} are two hyperwalks satisfying r=Stop flight Stop. To distinguish between the two occurrences of Stop in r, we rename the second occurrence into Stop'. In Stop flight Stop', Stop specifies the first stop (*vie2* for h_1 and *mun6* for h_2) and Stop' specifies the second stop (*par2* for h_1 and *par6* for h_2).

[5] Condition (2) did not exist in [AS92]. It is justified in Section 4.2.

Definition 1. Let r and r' be two hyperwalk expressions such that r' is obtained from r by renaming some types in r without changing the type definition: if t in r has been renamed into t' then $dom(t) = dom(t')$. Then r and r' have the same language: $\mathcal{L}(r) = \mathcal{L}(r')$. Let S be a set of hyperwalks satisfying *hwe* r. Renaming $\rho_{r'}(S)$ keeps S unchanged.

Selection allows one to evaluate Boolean functions (selection conditions) on hyperwalk labels. Applied to a set S of hyperwalks it returns the subset of hyperwalks in S whose labels satisfy the given conditions. We may distinguish in a regular expression r, "simple" subexpressions (without $*$) from "complex" subexpressions, i.e. those of the form $(s)^*$. If h is a hyperwalk in the instance of r, $I(r)$, then with a simple subexpression v in r is associated a unique component of h, denoted $h : v$, which is not necessarily a hyperwalk. To each complex subexpression $(s)^*$ of r corresponds a *set* of components in h, each of them satisfying s.

Example 3. In *hwe* $r =(\mathsf{Stop}_1\ \mathsf{flight})^*\ \mathsf{Stop}_2$, $(\mathsf{Stop}_1\ \mathsf{flight})^*$ is a complex subexpression and Stop_2 is a simple subexpression. For hyperwalk h={*vie3.LH347.mun3.AF456.par3*} in the instance of $(\mathsf{Stop}_1\ \mathsf{flight})^*\ \mathsf{Stop}_2$, the component $\mathsf{h{:}Stop}_2$ denotes *par3* and the component $\mathsf{h{:}}(\mathsf{Stop}_1\ \mathsf{flight})^*$ denotes the set {*vie3.LH347, mun3.AF456*}.

Definition 2. *Selection conditions* on *regular expressions* are defined as follows:

1. Let t be a type in a simple subexpression of r. A Boolean function $f(t)$ from $dom(t)$ into $\{true, false\}$ is a simple condition on r.
2. Let t and t' be two types in a simple subexpression of r. Then $t = t'$ is a simple condition on r.
3. If C is a condition on s, and $(s)^*$ is a complex subexpression of r, then $\exists C$ is a condition on r.
4. If C and C' are conditions on r, then so are $C \wedge C'$, $C \vee C'$ and `not` C.

Definition 3. Let C be a selection condition and S be a set of hyperwalks satisfying r: $S \subseteq I(r)$. The selection on S with condition C, denoted $\sigma_C(S)$, returns the hyperwalks in S satisfying C.

A hyperwalk h *satisfies* a selection condition C (C is true for h) if one of the following holds:

1. If C is simple, then it is evaluated in the straightforward way on the label of the corresponding component in h (as for tuples in the relational model).
2. If $C \equiv \exists D$ and D is a selection condition on s, then C is true iff the selection on the component of h, $h : (s)^*$, is not empty: h satisfies $\exists D$ iff $\sigma_D(h : (s)^*) \neq \emptyset$.
3. If $C \equiv D \wedge F$ then h satisfies C iff h satisfies D and h satisfies F.
4. If $C \equiv D \vee F$ then h satisfies C iff h satisfies D or h satisfies F.
5. If $C \equiv$ *not* D then h satisfies C iff h does not satisfy D.

Example 4. To get all hotels in Vienna, we apply $\sigma_{name(City)="Vienna"}$ on the hyperwalks satisfying $\mathsf{City\ addr\ Hotel}$. Let S be a set of journeys: hyperwalks satisfying $\mathsf{Journey\ first\ (Stop\ flight)^*\ Stop}$. To get all flight connections where at least one flight is on Air France, we apply $\sigma_{\exists company(flight)="Air\ France"}$ on S.

Projection: Informally speaking, the *projection* of a hyperwalk $h \models r$ on a *hwe* r' consists in keeping either a subset of the walks in h or subwalks of walks in h. For example, {*vie2.AF346.par2*} is a projection of h={*vie2.AF456.par2, par2.cdg.Paris*}. So is the set {*vie2*}, containing only the stop in Vienna.

The projection of a hyperwalk satisfying a hwe r on a hwe r' has to be itself a hyperwalk, i.e. a set of connected walks. One consequence of this is that the projection of each walk in a hyperwalk has to return a single walk (or the empty walk). As an example, consider the hwe $r = \mathsf{Journey\ first(Stop\ flight)^*Stop}$. The projection on $\mathsf{(Stop\ flight)^*Stop}$ returns for each walk w satisfying r exactly *one* subwalk with all stops in w connected by edges of type flight. But the same walk also satisfies the hyperwalk expression $r' = \mathsf{Journey\ first(Stop\ flight)^*Stop(flight\ Stop)^*}$ where the projection on $\mathsf{(Stop\ flight)^*Stop}$ is no more well defined since *any* subwalk of w starting in the first stop could be chosen as a result.

In [Ama94] a partial ordering ($\leq$) is defined on hyperwalk expressions: if $r' \leq r$, then r' is said to be a subexpression of r and each hyperwalk satisfying r can be projected on r'.

Definition 4. Let *hwe* r' be a subexpression of *hwe* r. Then the projection of a set of hyperwalks $S \subseteq I(r)$ on r', denoted $\pi_{r'}(S)$, is the set of the r'-components, denoted $h : r'$, of the hyperwalks h in S:

$$\pi_{r'}(S) = \{h : r' \mid h \in S\},$$

i.e. $\pi_{r'}(S)$ is a subset of the instance of r', $I(r')$.

Example 5. $\{la4.TWA567.par4.AF234.vie4.schwechat.Vienna.ring.Albertina\}$ is a hyperwalk satisfying hwe (Stop.flight)*Stop in City addr Monument. The projection on City addr Monument returns $\{Vienna.ring.Albertina\}$. The projection on (Stop.flight)*Stop returns $\{la4.TWA567.par4.AF234.vie4\}$.

Join:

Definition 5. Let $S \subseteq I(r)$, $S' \subseteq I(r')$ be two sets of hyperwalks. If $r + r'$ is a *hwe* (i.e. r and r' must share at least one node type) then the *join* of S and S', denoted $S \bowtie S'$, returns a subset of $I(r + r')$ each hyperwalk of which is the union of a hyperwalk in S and a hyperwalk in S':

$$S \bowtie S' = \{h \cup h' \mid h \in S, h' \in S', h \cup h' \models r + r'\}.$$

Note that in order the union of two hyperwalks $h \in S$ and $h' \in S'$ to be a hyperwalk satisfying $r + r'$, h and h' must share at least one node.

Example 6. Assume we have a set of cities and monuments $S \subseteq I($City addr Monument$)$. To get the hotels with a view on these monuments, we will join S with $I($Hotel view Monument$)$. The result satisfies the regular expression City addr Monument+Hotel view Monument. If S contains the hyperwalk $\{$*Paris.champ_de_mars-.Tour_Eiffel*$\}$ and Trocadero hotel has three free rooms with a view on the Eiffel tower, then $\{$*Paris-.champ_de_mars.Tour_Eiffel, Hotel_Trocadero.3.Tour_Eiffel*$\}$ is a hyperwalk in the result satisfying hwe City-addr Monument+Hotel view Monument

Concatenation of walks is defined in the usual way. Concatenation of two hyperwalk expressions $r = \sum_{1 \leq i \leq n} r_i$ and $r' = \sum_{1 \leq j \leq m} r'_j$ is defined as the sum of the concatenation of all r_i with all r'_j:

$$r \odot r' = \sum_{1 \leq i \leq n, 1 \leq j \leq m} r_i \odot r'_j$$

where $r_i \odot r'_j$ is empty if the ending node of r_i is not equal to the starting node of r'_j.

Definition 6. Let S and S' be two subsets of $I(r)$ and $I(r')$ respectively. If $r \odot r' \neq \epsilon$, then the *concatenation* of S and S', denoted $S \odot S'$, returns a set of hyperwalks in $I(r \odot r')$, each element of which is the concatenation of a hyperwalk in S with a hyperwalk in S'.

$$S \odot S' = \{h \circ h' \mid h \in S, h' \in S', h \circ h' \models r \odot r'\}$$

Example 7. Assume we have a set of cities and hotels $S \subseteq I($City addr Hotel$)$. To get the hotels with a view on the Secession building, we will concatenate S with $\sigma_{name(Monument)="Secession"}($Hotel view Monument$)$. The result satisfies regular expression City addr Hotel view Monument.

Several other, possibly useful, operations have not been studied. An example of them is *flatten*: a walk satisfying hwe (Stop flight)*Stop might be translated into the set of all its subwalks of length 1 satisfying the hwe Stop flight Stop.

4 Expressive Power of the Gram Algebra

In the model defined in Section 3 we did not make any assumption on the underlying type system. In this section we assume that nodes and edges have standard relational types and call *GramR* the corresponding subset of Gram. We shall first show that GramR without recursion has less expressive power than relational algebra (or first-order logic). Adding Kleene closure, then, suggests that GramR is included in the class of languages expressible in first-order logic with transitive closure (FO+TC). We will show that, in this respect, relational transitive closure is not sufficient to represent Kleene closure as it is used in Gram.

4.1 Relational Representation of Hyperwalks of Fixed Size

In a first step, we shall only consider hyperwalk expressions $r = \sum r_i$ without Kleene closure. Then all walks satisfying a walk expression $r_i = n_1 e_1 n_2 \ldots n_k$ are of fixed length $k - 1$. The representation of a GramR database by a relational database is straightforward:

1. with the instance $I(t)$ of each node type t, we associate a relation $rel(I(t))$ (with the same attribute names as in t) such that to each node $\nu \in I(t)$ corresponds a tuple $\lambda(\nu)$ in $rel(I(t))$.
2. with the instance $I(e)$ of each edge type e connecting nodes of type t and t' we associate a relation $rel(I(t\,e\,t')) \subseteq rel(I(t)) \times rel(I(e)) \times rel(I(t'))$ (with the same attribute names as in t, e, and t') which contains for each edge $\epsilon \in I(e)$ where $\psi(\epsilon) = (\nu, \nu')$ a tuple $\lambda(\nu) \times \lambda(\epsilon) \times \lambda(\nu')$.

Example 8. Figure 3 shows two relations. The first contains the labels of all cities ($rel(I(\mathsf{City}))$), and the second stores all cities and hotels with their addresses ($rel(I(\mathsf{City\,addr\,Hotel}))$).

$rel(I(\mathsf{City}))$		
c_name	**pop**	**country**
Vienna	1.7	Austria
Paris	3	France
Los Angeles	9	USA
...	...	...

$rel(I(\mathsf{City\,addr\,Hotel}))$					
c_name	**pop**	**country**	**addr**	**h_name**	**rooms**
Vienna	1.7	Austria	ring	Marriott	146
Vienna	1.7	Austria	oper	Sacher	34
Paris	3	France	st.jacques	Pullman	87
...	...	...	...	...	...

Figure 3: Two relations for City and City addr Hotel

The next step consists in constructing a relation with tuples representing walks and hyperwalks without Kleene closure. Let $rel(S)$ denote the relation representing hyperwalks in $S \subseteq I(r)$.

3. Concatenation ($\odot$): The hyperwalks in $S \odot S' \in I(r \odot r')$ are represented by the natural join $rel(S) \bowtie rel(S')$[6] where the common attributes of $rel(S)$ and $rel(S')$ correspond to the "last" (node) attributes in r and the "first" (node) attributes in r'.
4. Join ($+, \bowtie$): The hyperwalks in $S \bowtie S' \in I(r+r')$ are represented by the natural join $rel(S) \bowtie rel(S')$.

4.2 GramR without Kleene closure

Theorem 7. *GramR without Kleene closure (GramR$^-$) is included in first-order logic (FO).*

Proof. (Outline) Let $S \subseteq I(r)$ and $S' \subseteq I(r')$ be two sets of hyperwalks. With each operation op in the Gram algebra we associate a relational operation op': for unary operations we show that $rel(op(S))) = op'(rel(S))$ and for binary operations that $rel(op(S, S')) = op'(rel(S), rel(S'))$.

Selection: Since there is no Kleene closure in r, a selection condition is always constructed from simple conditions (with $\wedge, \vee, not$). GramR$^-$ assumes relational types, which means that any simple condition C on S (Definition 2.1, 2.2 and 2.4) is a relational selection condition on $rel(S)$. Then it is easy to check that

$$rel(\sigma_C(S)) = \sigma_C(rel(S)).$$

Renaming is straightforward since, as relational renaming, renaming S keeps S unchanged. In [AS92] renaming had a different semantics. Renaming of r might select a subset of the hyperwalks satisfying r. As an example take $r = \mathsf{Stop\,flight\,Stop} + \mathsf{Stop\,in\,City}$. Although there are two occurrences of Stop in the first term, this was an acceptable hwe for the model in [AS92]: $I(r)$ included hyperwalks where either the departure stop or the destination stop was connected to a city. Then by renaming – say the first occurrence of Stop – it was possible to select the subset of $I(r)$ only with the hyperwalks whose destination stop is connected to a city. In order to guarantee GramR$^-$ to be expressible in FO, we have forbidden such hyperwalk expressions (Condition 2 in the definition of hyperwalk expressions, Section 3) for which the hyperwalk "structure" is indeterministic: only $r_1 = \mathsf{Stop_1\,flight\,Stop} + \mathsf{Stop\,in\,City}$ and $r_2 = \mathsf{Stop\,flight\,Stop_2} +$

[6] Wlg., we assume node and edge types with only one attribute.

Stop in City are now valid expressions. This has, of course, for a consequence that renaming keeps the set of hyperwalks unchanged.

Projection: Similarly to selection, one might show for projection that

$$rel(\pi_{r'}(S)) = \pi_X(rel(S))$$

where r' is a subexpression of r and X is the set of attributes corresponding to the node and edge types in r'. But the opposite is not true. Opposed to Gram projection, where the resulting schema subgraph has always to be connected, relational projection allows one to project on *any* subset Y of attributes in $rel(S)$. For example by relational projection it is possible to project only on the hotels and monuments in relation $rel(I(\text{City addr Hotel} + \text{City addr Monument}))$, which is not allowed in Gram (the corresponding schema subgraph is not connected anymore).

By lack of space the proofs for the binary operations are omitted. Clearly, GramR$^-$ is strictly less expressive than FO since hyperwalks are always connected: relational projection is more powerful than Gram projection and there is no equivalent to relational cartesian product.

4.3 GramR with Kleene closure

A set of paths of fixed length n can always be represented by a relation of arity $2*n+1$. This is not the case for sets of paths of *variable length*. Indeed, let S be a set of variable length walks satisfying hwe (Stop flight)*Stop. Edges of a walk w in S are represented by tuples in $R = rel(I(\text{Stop flight Stop}))$. Let $TC(R)$ be a relation with two attributes of type Stop such that there is a tuple $[\lambda(\texttt{s}), \lambda(\texttt{s'})]$ in $TC(R)$ iff s is a stop connected to stop s': $TC(R)$ is the transitive closure of $\pi_{\$1,\$3}(rel(I(R)))$.

Clearly, a walk in S can be represented by a subset of R, or a subset of $TC(R)$, but there is no relation representing S itself. Therefore, transitive closure is not sufficient for representing sets of variable length paths (ordered sequences) such as modeled by Gram. In that respect, GramR has more expressive power than FO+TC and thus than GraphLog: in [CM90] the expressive power of GraphLog was shown to be equivalent to that of stratified linear Datalog, first-order logic plus transitive closure, and non-deterministic logarithmic space (assuming ordering on the domain).

This increase in the expressive power of GramR should also increase its complexity. However, observe that selection (Section 2) is linear in the size of the set of hyperwalks[7].

5 Query Navigation

We shall describe several basic mechanisms for database navigation. Gram is used as the underlying model and the example application is guided hypertext navigation.

5.1 Simple Navigation

We assume a window based hypertext interface where several documents can be displayed at one time. Without querying capabilities, simple navigation consists in clicking on a $\boxed{\text{Button}}$ in a displayed document *source*, which results in following a link (*source*, *target*), i.e. in the display of the document *target*. This corresponds, of course, to a projection on the target node in the relational implementation of the link.

From now on, the set of displayed documents will be called the current *user state* U [Tom89]. The source document (node) of the followed link will be called the *active* document (node) in order to distinguish it from the other documents in U.

5.2 Virtual Link or Short-cut

A *virtual link* or *short-cut* defined by a query Q and denoted by $link(Q)$, is the set of node pairs connected by paths in Q:

$$link(Q) = \{(n_1, n_k) \mid n_1 e_1 ... e_{k-1} n_k \in Q\}.$$

[7] This fact allowed the implementation of selection by sequential filtering in VERSO N1NF relations represented as sets of sequences (N1NF tuples) satisfying a regular expression [BS80, AB86].

Any such nodes n_1 and n_k are said to be connected by the virtual link $link(Q)$. While Q brings complete information on its paths, $link(Q)$ only keeps the paths' end nodes. A virtual link can be followed from some active node $n \in U$ by the operation

$$goto(n, Q) = \{n' \mid (n, n') \in link(Q)\},$$

i.e. $goto(n, Q)$ opens all documents reached from n by following a path in Q.

5.3 Ad-hoc Query for Multiple Steps Navigation

The simplest query assisted navigation is to interactively type a query Q when displaying a document d (d is the active document). The effect of the interpretation of Q is the display of documents in $goto(d, Q)$. Instead of navigating through the database step by step (document by document), by ad-hoc querying the user directly reaches documents in $goto(d, Q)$.

Example 9. In order to obtain all cities visited during the journey *Bridges&Canals* it is possible to navigate through each stop and enter the city description by following the link of type in. A better solution is to use the query $Q = \sigma_{name(Journey)=\text{"}Bridges\&Canals\text{"}}$ Journey first(Stop flight)*Stop in City which has for a result the short-cut $\{(Bridges\&Canals, Venice), (Bridges\&Canals, Amsterdam), (Bridges\&Canals, St.Petersburg)\}$. This example also shows how virtual links prevent unnecessary navigation by following different paths to the same document: although there might be two stops in Amsterdam, it appears only once in the set of virtual links.

Such a short-cut might as well be stored with a new button $\boxed{\rightarrow, Q}$ in document d. Clicking on this button triggers the interpretation of Q and opens documents in $goto(d, Q)$. "Hardwired" incremental updates are one of the limits of current hypertext systems. This short-cut mechanism introduces flexibility and allows the control of the database size: it avoids the unnecessary creation of actual edges between documents. Note also that (1) $goto(d, Q)$ might be precomputed for faster navigation, but implies recalculation when the hypertext graph is modified, and (2) this mechanism can be implemented in a relational model by a view mechanism. A multitude of hypertext systems attach scripts (procedural programs) to buttons [RS92, Goo87]. Short-cuts as defined above can then be implemented as scripts in such systems.

A higher-level user interface for multiple steps navigation could be implemented by providing the user with a *(schema) browser* of the hypertext schema graph. By selecting nodes and links in the schema browser, he/she can follow a short-cut $link(Q)$, where Q is the query that corresponds to the graphical selection.

Example 10. Instead of following paths satisfying hwe Journey first(Stop flight)*Stop in City in order to obtain all cities in a journey, the reader might first select a document about a journey, say *Bridges&Canals*, and then describe the path to be followed by a graphical selection of the nodes Stop and City in the schema graph.

5.4 Navigation Space Restriction

Navigation, even when query assisted, becomes extremely complex when the number of displayed nodes and edges passes some limit. Navigation space restriction is then necessary. Several mechanisms have been proposed in the hypertext literature for hypertext systems [Zel89, SF89]. We suggest to restrict navigation to paths defined by a query Q. Then the user navigates step by step but only in the predefined space: in each node only buttons corresponding to the paths of Q are active, the others being disabled. As for short-cuts this mechanism can be implemented either by storing Q with a button $\boxed{\otimes, Q}$ or by ad-hoc querying.

Example 11. The query $Q = \sigma_{country(City)=\text{"}France\text{"}}$(Journey first(Stop flight)*Stop in City) restricts navigation to French cities. It is not possible to follow a link to another city.
The query $Q = \sigma_{value(date)=8/3/1993}$(City addr Monument + Guide date Monument) allows the user to visit only monuments where a guide is available on August 3, 1993.

5.5 Reverse Links and History

Navigating through a database implies an asymmetric view of data: data is accessed through directed paths in a graph. However this asymmetry is only apparent. We can always compute a set of nodes from which a given node can be reached through paths satisfying query Q. This corresponds to the projection of the paths in Q on the *source* nodes of the paths. Therefore reverse links satisfying a query Q may be defined by

$$comingfrom(n, Q) = \{n' \mid (n', n) \in link(Q)\}.$$

Example 12. From Marriott hotel one can obtain general touristic information about Vienna by activating a button $\boxed{\leftarrow, Q}$ that follows the paths in query Q =City addr Hotel in the reverse direction.

Hypertext systems such as Guide[Bro87] and HyperCard[Goo87] allow to return to some previously visited document by using a special "back" button $\boxed{\hookleftarrow}$. NoteCards[Fos89] displays the user history in some form such that a previous document may be identified and selected visually. Reverse links as defined above ($comingfrom$) are insufficient for implementing history mechanisms. For example, after having taken a flight from a stop in Los Angeles to a stop in Paris, the activation of the button $\boxed{\leftarrow,\text{Stop flight Stop in City}}$ in Paris does not return only to the stop in Los Angeles but to *all* stops which are the departure of a flight to Paris.

We shall implement user history as follows. Each time a link is followed, the source document (active document) is pushed onto a stack $\mathcal{H}$[8]. Upon the display of a document, thanks to the above stack mechanism, the user can navigate back to the previous document ($pop(\mathcal{H})$) or to the first one ($empty(\mathcal{H})$). Combined with queries, he/she might then implement more sophisticated reverse strategies: instead of following the history step by step (button $\boxed{\hookleftarrow}$), he/she might pop a whole path p returned by a query Q: let p be a linear path connecting documents $d_1, d_2, \ldots d_k$ in the same order as they were pushed onto $\mathcal{H}$. We assume that d_k is the active document and has already been pushed on $\mathcal{H}$. Then p can be popped from $\mathcal{H}$ and d_1 becomes the active document: $pop(\mathcal{H}, p)$. As in Guide and HyperCard, we can use a back button and attach to it a query Q: $\boxed{\hookleftarrow, Q}$. By clicking on this button, the starting document of the paths p in Q that can be popped from H will be displayed.

Example 13.) A client reading a journey description (following a walk satisfying hwe Journey first(Stop-flight)*Stop in City addr Hotel has obtained some information about a hotel by following two links of type in and addr. In order, then, to resume the journey, he might click on a button $\boxed{\hookleftarrow,\text{Stop in City addr Hotel}}$.

6 Database Reutilization

We shall show in this section how to reutilize existing hypertexts for building new ones.

6.1 Travel Agency Application Revisited

In the travel agency example (Figure 1) Stop documents are only justified, because a city c can belong to an unknown number of journeys and we do not want to add a new anchor to the document c, each time a journey j goes from c to another city c'. Indeed, with the schema in Figure 1, we distinguish between the city's view as a flight stop (document of type Stop) from the city to be visited (document of type City). A new flight connection is established by connecting two new Stop documents linked to the corresponding documents of type City. Observe, however, that with this schema and without a mechanism such as that defined in Section 5.5 (Example 13), once the user enters a city, he/she is stuck into it and cannot get back to the corresponding Stop document by simple navigation.

In Figure 4 the return links are not actual links (for the same reason as above, there is an unknown number of such links leaving a city) but they are implemented by the mechanism of Section 5.5. Each time a user adds a new trip, he/she has first to create new Stop documents and corresponding flight links plus new links between stops and cities. This allows the reutilization of the existing touristic hypertext without increasing its complexity.

[8] A more complex mechanism could be implemented by pushing the whole user state onto the stack. When considering all documents in a state at the same level of a tree, a history mechanism would then be defined as some form of tree traversal.

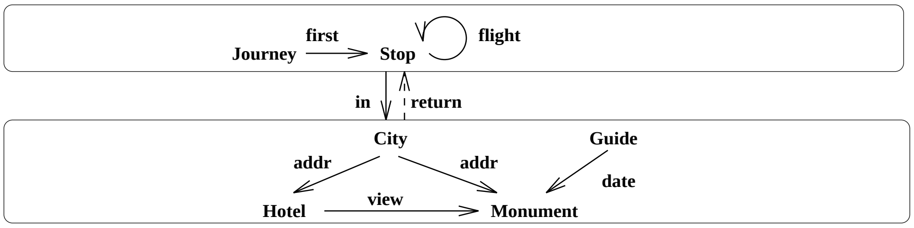

Figure 4: Journey and Touristic Information Hypertexts

6.2 Call or Zoom

The following mechanism allows a better connection and structuralization between separate hypertexts. In particular, it becomes possible to "call" a hypertext H from some document in another hypertext H'. Looking at the schema in Figure 4, we can think of one hypertext on journeys reutilizing one on touristic information. Each document of type Stop now contains a button $\boxed{\downarrow}$ that can be activated in order to enter the corresponding touristic hypertext of the stop: links of type in are not necessary anymore.

Zooming Function: Let S be a database schema and t be a node type not in S. Then $zoom : I(t) \rightarrow \mathcal{I}(\mathcal{S})$ associates with a node of type t a hypertext with schema $\mathcal{S}$. $\mathcal{I}(\mathcal{S})$ denotes the set of hypertexts with schema $\mathcal{S}$.

Clicking on button $\boxed{\downarrow}$ in stop s activates the zoom, i.e. opens the hypertext $zoom(s)$. Some entry points, i.e. documents in $zoom(s)$, have of course to be specified. Instead of following a path from some stop s into $zoom(s)$ and navigate, say from a city to its hotels, we might directly apply a query on $zoom(s)$: the activation of the zooming button $\boxed{\downarrow, Q}$ enters hypertext $H = zoom(s)$ and applies query Q on H. Afterwards we may return to stop s by clicking on button $\boxed{\uparrow}$.

However, a query might involve both, a path p on journeys as well as a path p' on touristic information *embedded* in p.

Embedded Paths: Let Q be a set of paths satisfying hwe r. Let t be a type in r and $zoom : I(t) \rightarrow \mathcal{I}(\mathcal{S})$ be a zooming function. Let Q' be a query on hypertexts with schema S. Then we can enter from a path $p \in Q$ into a path $p' \in Q'$ if p contains a document d such that p' is a path in $zoom(d)$:

$$embed(Q, Q') = \{(p, p') \mid p \in Q \text{ and } \exists d \in p : \tau(d) = t \wedge p' \in Q'(zoom(d))\}$$

defines all pairs of paths (p, p') such that p' can be entered from path p.

For representing such pairs of paths, we will use the following rewriting rule where a dot notation is used to indicate the document to be zoomed:

Rewriting: Let $(p, p') \in embed(Q, Q')$ be a pair of paths such that p satisfies $r = u\,t\,v$ and p' satisfies r'. Then (p, p') is said to satisfy hwe $s = ut.(r')v$ (if there is no ambiguity, the parentheses will be omitted).

Example 14. "Get the journey where it is possible to visit a monument with the guide named Lydie" can be answered as follows:

```
SELECT *
  FROM Journey first (Stop flight)* Stop.Guide date Monument
 WHERE Guide.name="Lydie" AND
       Stop.date=date.value
```

Note that we compare the date of a Stop document in the journey hypertext with the value of the date link in the touristic information hypertext. An alternate syntax a la O_2SQL [BCD89] for such queries could be the following:

```
SELECT *
  FROM p  IN Journey first(Stop flight)*Stop,
       p' IN p.Stop.Guide date Monument
 WHERE p.Stop.date=p'.date.value AND
       p'.Guide.name="Lydie"
```

Here $p \in I(\mathsf{Journey\ first\ (Stop\ flight)^*Stop})$ is a path in the journey hypertext and $p' \in I(\mathsf{Guide\ date\ Monument})$ is a path in the touristic information hypertext that can be entered from the last stop in p. Note that the dot notation is used for accessing attribute values as well as paths of a hypertext reached by zooming. Paths are "complex attribute values".

Example 15. "Give me the journeys to Vienna with a stop in Paris for visiting monuments with Lydie":

```
SELECT *
  FROM p  IN Journey first(Stop flight)*Stop1(flight Stop)*flight Stop2,
       p1 IN p.Stop1.Guide date Monument+City addr Monument
       p2 IN p.Stop2.City
 WHERE p.Stop.date=p'.date.value AND
       p1.City.name="Paris" AND
       p1.Guide.name="Lydie" AND
       p2.City.name="Vienna"
```

7 Conclusion and Future Work

We have studied in this paper the suitability of the Gram query language for complex navigation through databases. We discussed its expressive power and compared it to first-order logic plus transitive closure (FO+TC). We have shown that Gram is not included in FO+TC, mainly because transitive closure is insufficient for representing sets of variable length paths. On the other hand, Gram without recursion has less expressive power than first-order logic. One main reason for this is that the Gram algebra manipulates *connected* subgraphs, which imposes some restrictions on Gram projection and prohibits an operation corresponding to relational cartesian product.

Several mechanisms for high-level database navigation have been presented and illustrated with Gram. The paths in a query Q can be used as *short-cuts* allowing multiple-steps navigation. This mechanism introduces flexibility and allows a control of the database size. Similar to views in relational databases, it avoids the creation of actual edges between nodes. By restricting user *navigation space* to paths of a query Q, browsing in complex graphs becomes easier, especially for casual users. A *user history* mechanism was defined for coming back to previous states. The user might, then, not only return to previous documents step by step, but also by following a path in the history. Finally, *zooming* was introduced as a useful information structuring. In particular, it allows a better reutilization of existing databases and the modeling of hierarchical relationships. This is especially helpful for designing transportation networks in which a node of the network can in turn be viewed as (zoomed in) a network itself. [Güt91] proposes a data model for modeling such transportation networks.

For validating the above mechanisms we have chosen a hypertext application. The HyperPATH hypertext system [RS92] was first integrated [ACS93] with the O_2 DBMS [Deu89]. The next step will be the extension of HyperPATH/O_2 to include query capabilities and the use of the HyperPATH script language and of the O_2SQL query language [BCD89] for implementing navigation strategies.

Acknowledgments We wish to thank S. Abiteboul, V. Christophides, S. Grumbach, V. Vianu and A.M. Vercoustre for helpful discussions.

References

[AB86] S. Abiteboul and N. Bidoit. Non first normal form relations: An algebra allowing data restructuring. *Journal of Computer and System Sciences*, 33(3):361–393, December 1986.

[ACS93] B. Amann, V. Christophides, and M. Scholl. HyperPATH/O2: Integrating hypermedia systems with object-oriented database systems. In *Proc. of the 4th Int. Conf. on Data and Expert Systems Applications (DEXA93)*, Prague, Czech Republic, September 1993. Paru également dans *IXèmes Journées Bases de Données Avancées*, Toulouse, France, Septembre 1993.

[AH87] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Trans. on Database Systems*, 12(4):525–565, December 1987.

[Ama94] B. Amann. *Hypertexts and Databases*. PhD thesis, CNAM, Paris, 1994. in preparation.

[AS92] B. Amann and M. Scholl. Gram: A graph data model and query language. In *Proc. of the 4th ACM Conf. on Hypertext and Hypermedia (ECHT'92)*, Milano, Italy, December 1992. Paru également dans *VIIIèmes Journées Bases de Données Avancées*, Trégastel, France, Septembre 1992.

[BCD89] F. Bancilhon, S. Cluet, and C. Delobel. A query language for the O_2 object-oriented database system. In *Proc. of the 2nd Int. Workshop on Database Programming Languages (DBPL'89)*, 1989.

[BK90] C. Beeri and Y. Kornatzky. A logical query language for hypertext systems. In *Proc. of the 1st European Conf. on Hypertext and Hypermedia (ECHT'90)*, pages 67–80, Versailles, November 1990.

[Bro87] P.J. Brown. Turning ideas into products: The Guide system. In *Proc. of Hypertext '87*, pages 33–40, November 1987.

[BRS90] J. Biskup, U. Räsch, and H. Stiefeling. An extension of SQL for querying graph relations. *Computer Languages*, 15(2):65–82, 1990.

[BS80] F. Bancilhon and M. Scholl. Design of a back end processor for a data base machine. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Los Angeles, 1980.

[Che76] P.P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Trans. on Database Systems*, 1(1):9–36, 1976.

[CM89] M.P. Consens and A.O. Mendelzon. Expressing structural hypertext queries in GraphLog. In *Proc. of Hypertext '89*, pages 269–292, November 1989.

[CM90] M.P. Consens and A.O. Mendelzon. GraphLog: a visual fromalism for real life recursion. In *Proc. of the ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pages 404–416, Nashville, Tennessee, 1990.

[CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. G+: Recursive queries without recursion. In L. Kerschberg, editor, *Expert Database Systems*, pages 645–666. Benjamin/Cummings, 1988.

[COD71] CODASYL. Data Base Task Group April 71 Report. ACM, New York, 1971.

[Con87] B. Conklin. Hypertext: an introduction and survey. *IEEE Computer*, 20(19):17–41, September 1987.

[Deu89] O. Deux. The Story of O_2. *IEEE Trans. on Knowledge and Data Engineering*, 2(1), March 1989.

[Fos89] C.L. Foss. Tools for reading and browsing hypertext. *Information Processing & Management*, 25(4):407–418, 1989.

[Goo87] D. Goodman. *The Complete HyperCard Handbook*. Bantam Books, inc., N.Y., USA, September 1987.

[GPG90] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object model for database end-user interfaces. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 24–33, May 1990.

[Güt91] R.H. Güting. Extending a spatial database system by graphs and object class hierarchies. In G. Gambosi, H. Six, and M. Scholl, editors, *Proc. of the Int. Workshop on Database Management Systems for Geographical Applications*, Capri, May 1991.

[Hal88] F.G. Halasz. Reflections on NoteCards: Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, 31(7):836–852, July 1988.

[Har71] F. Harary. *Graph Theory*. Addison Wesley Series in Mathematics, 1971.

[HK87] R. Hull and R. King. Semantic data modeling: Survey, applications and research issues. *Theoretical Computer Science*, 19(3), September 1987.

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–402, 1992.

[NN91] J. Nanard and M. Nanard. Using structured types to incorporate knowledge in hypertext. In *Proc. of Hypertext '91*, pages 329–343, December 1991.

[RS92] A. Rizk and L. Sauter. Multicard: An open hypermedia system. In *Proc. of the 4th ACM Conf. on Hypertext and Hypermedia (ECHT'92)*, December 1992.

[SF89] P.D. Stotts and R. Furuta. Petri-net-based hypertext: Document structure with browsing semantics. *ACM Trans. on Information Systems*, 7(1):3–29, January 1989.

[Tom89] F.W. Tompa. A data model for flexible hypertext database systems. *ACM Trans. on Information Systems*, 7(1):85–100, July 1989.

[Zel89] P.T. Zellweger. Scripted documents: A hypermedia path mechanism. In *Proc. of Hypertext '89*, November 1989.