Languages and Tools to Specify Hypertext Views on Databases[1]

G. Falquet, J. Guyot, L. Nerima
Centre universitaire d'informatique, University of Geneva, Switzerland
falquet | guyot | nerima @cui.unige.ch
http://cuiwww.unige.ch/db-research/hyperviews

Abstract. We present a declarative language for the construction of hypertext views on databases. The language is based on an object-oriented data model and a simple hypertext model with reference and inclusion links. A hypertext view specification consists in a collection of parameterized node schemes which specify how to construct node and link instances from the database contents. We show how this language can express different issues in hypertext view design. These include: the direct mapping of objects to nodes; the construction of complex nodes based on sets of objects; the representation of polymorphic sets of objects; and the representation of tree and graph structures. We have defined sublanguages corresponding to particular database models (relational, semantic, object-oriented) and implemented tools to generate Web views for these database models.

1 Introduction

The hypertext navigation paradigm [4] has proven highly efficient for easily sharing and accessing information without having to learn any specific query language or having to know the information storage structure (see for example most of the on-line help systems of recent software products or the World-Wide-Web global information system 4.). Moreover, database systems become universally used for storing, retrieving, and processing large amounts of data in an efficient and secure way. This lead to the development of so-called "database publishing" tools which make the content of a database accessible through a Web interface or to data browsing tools.

A hypertext view is a (virtual) hypertext derived from the database relations or classes. From the view concept it borrows the idea of a view definition language that specifies how to compute the view. But instead of producing a derived database element (class) it produces nodes and links that form a hypertext. The aim of a hypertext view are:

- to represent a part of the content of a database and
- to replace traditional query operations (selections, joins, projections) by a reasonable number of navigation steps in the hypertext.

The last goal implies that the hypertext structure must be carefully designed. For instance, it is not sufficient to simply map each database object onto a hypertext node and

1. This work was partially supported by the Swiss National Science Foundation, grant no. 21-45791.95

each object relationship to a hypertext link. We have defined a language and implemented tools to simplify and study the task of producing hypertext views.

The rest of this paper is organized as follows: In the next section we present our hypertext view definition language for object-oriented databases. In section 3 we analyze several issues that arise when designing hypertext views and we show how they can be handled with our language. In section 4 we present the formal syntax and semantics of the language and in section 5 we compare our approach with others.

2 The Models and the Hypertext View Definition Language

Building hypertext views consists essentially in mapping database objects to hypertext objects. We chose simple database and hypertext models to base the view definition language on. These models can then be easily mapped to other database and hypertext models to build concrete tools. For instance, we have implemented concrete tools for viewing relational databases and O2 databases on the Web.

2.1 The Database Model

The database model is an object-based model which is a subset of the O2 database model [11][1]. In this model each object has an identity (oid) and a value. We only consider objects whose value are tuples [$name_1$: $value_1$, $name_2$: $value_2$, …, $name_n$: $value_n$] where each $name_i$ is an attribute name and each $value_i$ is either: an atomic value (of type `integer`, `boolean`, `float`, `string`, etc.); or a reference to another object; or a collection of references (set, list, bag). A *database schema* is a set of class definitions which are composed of a class name and a type (we do not consider methods) and an inheritance relationship. The type of a class constrains the values of its objects. A *database instance* is a set of objects and a set of named collections. Each object is an instance of a class and its value belongs to its class type. A named collection is a set (or list) of objects which are instances of a class or of one of its subclasses.

2.2 The hypertext model

We consider a simple hypertext model whose structural part is composed of nodes, anchors, and links.

Each *node* has a unique identity and a content.

The *content* of a node is a sequence of *elements* which may be character strings, images, etc.

An *anchor* is an element or a sequence of elements within the content of a node, it serves as a starting or ending point of a link.

A *link* is defined by its starting and ending anchors and by its category which is either ‘reference’ or ‘inclusion’. Reference links are intended to create a navigation structure within the nodes. Inclusions links are intended to create nested structures that represent complex contents (structured documents).

This model can easily be mapped to the Web model. The only problem comes from inclusion links because, in HTML, the <A HREF...> tag corresponds to reference links and there are no inclusion links (except for images, with the IMG tag). The approach we took when creating tools for the Web, was to represent in a single HTML page the content of a node and of all its subnodes and to use embedded list tags to show the inclusion structure (Fig. 2 shows an example).

2.3 The Hypertext View Definition Language

A hyperview specification consists of set of node schemes which specify the collection from which the node's content is to be drawn; the selection and ordering criteria; the elements that form the content; and links to other nodes. A node definition takes the following form:

```
node <node-name> [ <parameters> ] is
       <field-list>
  from <collection>
  selected by <expression>
  ordered by <expression>
```

Each `<field>` of the `<field-list>` can be either a literal constant (string, integer, etc.) or an attribute name. Fields may contain presentation functions (bold(), italic(), break(), paragraph(), heading1(), etc.) which generate markup tags for the target hypertext system.

Content of a node. The content of a node is based on a collection specified in the `from <collection>` clause. The sequence of fields specifies how to construct the elements that represent every selected object. For example, let `employeesOfDept` be defined as:

```
node employeesOfDept[d: Department] is
  " No: ", bold(no), " => " ,
  bold(name), break(),
  " Hire date: ", hire_date
from EMP selected by dept = d
     ordered by emp_name
```

The content of a node instance `employeesOfDept[sales]` is obtained by

- selecting all the objects in the collection EMP which have the value `sales` for their attribute `dept`
- ordering these objects according to the values of attribute `name`
- for each object creating the sequence of elements corresponding to: the string `"No: "`, the value of attribute `no` (surrounded by <B> and </B> tags), the string `"=>"`, etc.

It will appear as shown in Fig. 1[1]

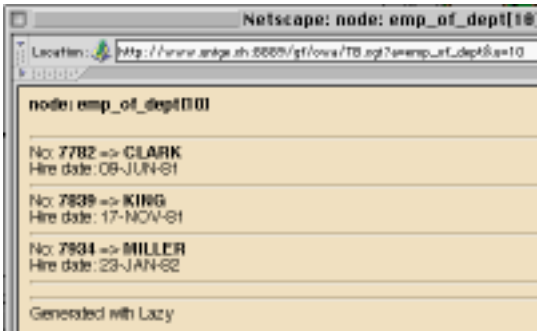

Figure 1. A simple node instance

Reference Links. Links are specified through the `href` statement. The starting point (anchor) of a link is always a field. The anchor text will form an active element in the starting node which can trigger the navigation to the referenced node. A link specification refers to a node through its schema name and a list of parameter values. It is an expression of the form:

```
href <schema_name> [ <value>, ... ] <field>
```

For instance, consider the following node definition:

```
node dept_in[loc: String] is
  no, ": ", bold (name), " ",
  href employeesOfDept [self] " Employees: "
  ...
from DEPT selected by location = loc
  ordered by no
```

(Note: the pseudo variable *self* iterates over the sequence of selected objects).

The representation of each selected department `d` will have an anchor text "Employees:" that is the starting point of a link to the node `employeesOfDept[d]`.

Inclusion Links. An inclusion link between two nodes determines a compound-component relationship between these nodes. The target node is to be considered as a sub-node of the source node of the link. This fact should normally be taken into account by the hypertext interface system which should present inclusion links in a particular way (generally by including the sub-node contents within the node presentation). Inclusion

---

1. Illustrations are snapshots of Web pages dynamically generated with the LAZY system. A description of this implementation can be found in http://cui-www.unige.ch/db-research/hyperviews/

links are particularly useful to create multi-level hierarchical nodes to represent complex entities.

The figure below shows an instance of a node `dept_in2` which has the same definition as `dept_in` except for the reference link which is replaced by the inclusion link `include employeesOfDept[self] " Employees:"`

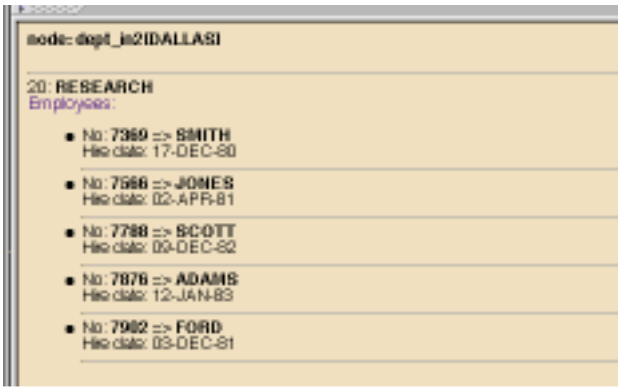

Figure 2. A node with an inclusion link

The content of an included node may depend on a distance parameter. By definition, a node is at distance 0 from itself and a node included in a node at distance i is at distance i + 1. A visibility can be associated with each field of a node schema to indicate the maximum distance from which this field is visible. The content of a node at distance d is composed of all the fields with a visibility greater or equal to d. As we will see in the next section, this mechanism has useful applications such as:

- avoiding infinite inclusion structures (at a given maximum distance all included nodes become empty);
- including summaries or outlines of nodes to reduce the number of navigation steps.

3 Designing Hypertext Views

Hypertext design is a complex problem, which has attracted many research (see [21] for instance). More recently, methodologies have been developed to design hypertext views of databases [2][9][8]. In the case of hypertext views of databases the design can take advantage of the data semantics expressed in the database schema but, as mentioned in [2], the distance between the database schema and the hypertext structure is great. For instance, a good database structure should minimize data redundancy to avoid update anomalies. On the contrary, redundancy may help the hypertext user by reducing the number of navigation steps to reach some information. In database design, the length of logical access paths is not important since the database is generally accessed through application programs or high level interfaces (e.g. forms). In hypertexts, since the basic

action is the navigation step, the number of links to traverse is an important criterion to determine the usability of the system.

In this section we will show how our language can be used in several design cases.

3.1 Direct Object Mapping

The most straightforward way to map database entities to a hypertext structure consists in taking each object o of the database to create a hypertext node node(o). In this situation, the attributes with an atomic value form the content of the node. An attribute that refers to another object o' gives rise to a reference link to node(o'). A multivalued attribute generates a link to an index node which in turn points to the nodes representing the individual objects. Thus the structure of the hypertext view is isomorphic (in terms of graph) to the database structure.

The direct object mapping can be specified in the following way: for each collection C of type c with attributes a_1, …, a_k (atomic), s_1: D_1, …, s_r: D_r (single valued references), and m_1: set(E_1), …, m_t : set(E_t) (multi-valued references) define a node schema:

```
node C [me: c] is
  a1, …, ak
  href D1[s1],
  …,
  href Dr[sr]
  include E1Index[m1],
  …,
  include EtIndex[mt]
from C selected by self = me ordered by 1
```

Each instance of this node schema represents a single object of the collection *C*. Single valued attributes yield references to the node representing the referred object. Multivalued attributes yield inclusions of index nodes which point to all the referred objects. The index nodes have the following schema:

```
node EiIndex [e: set Ei] is
  href Ei[self]
from e
```

3.2 Mapping Homogeneous Sets of Objects to Nodes

In order to reduce the number of navigation steps, the hypertext designer may wish to present several (or all) objects of a collection in a single node. This type of presentation is directly supported by the language since each node represents a subset of a collection (from <collection>) specified by a predicate (selected by <predicate>).

For instance, a node instance `employeesOfDept[s]` (defined in 2.3) represents all the employees of department `s (from EMP selected by d = dept)`. Thus a ref-

erence link to `employeesOfDept[s]` can lead in a single step from a node representing department `s` to a node representing all its employees.

3.3 Derived Links

Since the selection predicate of a node is not limited to reference attributes, it is possible to create new (computed) links that do not appear explicitly in the database schema. This is shown on the following example:

```
node CitiesNear[c: Coordinates] is
  name, ...
from CITIES selected by
   location.distance(c) < 100

node City is
  name, population, ...
  href CitiesNear[location] "nearby cities"
from CITIES ...
```

The reference link from `City` to `CitiesNear` leads from a node representing a city c to a node representing all the cities which are less than 100km from c.

3.4 Mapping Sets of Heterogeneous Objects to Nodes

In the previous section we have shown how to define nodes by selecting sets of objects from the same collection. We now consider aggregation nodes which are made of objects coming from different collections. In this case grouping occurs along the inter-collection axis instead of the within-collection axis.

This type of node construction is particularly useful to reconstruct complex entities which have been decomposed and represented as several interrelated objects stored in different collections. It is natural to group all the objects that represent a complex entity in a single hypertext node. The node's content can be organized hierarchically to reflect the structural composition of the complex object. The main advantage of this mapping lies in its compact presentation of related information, thus avoiding navigation operations among the different components of the complex entity.

To create aggregation nodes we use inclusion links that point to subnodes. The following example shows the construction of a complex node with a nested structure to represent courses stored in a university database.

Database schema:

```
class Course(
  code : String,
  title : String,
  credits : Int,
  description : String,
  prerequisites : set(Course) )
```

```
class Offering(
     code : String,
     course : Course,
     semester : String
 )
```

```
class Professor(
  name : String,
  ...
)
```

```
class Teaching(
     offering : Offering,
     professor : Professor
)
```

```
COURSES : set(Course); OFFERINGS : set(Offering);
TEACHINGS : set(Teaching); PROFESSORS : set(Professor);
```

Node schemes:

```
node Course [c : Course]  is
   heading1(code, " * ", title),
      "credits: ", credits,
   heading3("Description"),
      description,
   heading3("Prerequisites"),
      include Prereqs [prerequisites] ,
   heading3("Offerings"),
      include Offerings[self]
from COURSES
   selected by self = c ordered by code

node Prereqs [pre : set Course]
      list_type:  enumeration(" ")
   is
      href course[self] code, " (", title, ")"
from pre

node Offerings[c : course]
      list_type: definition
   is
      bold(code), " (", semester, ") ",
      include  Teaching[self]
from OFFERINGS
      selected by course = c order by code

node Teachings[o : Offering]
      list_type: enumeration(" ")
   is
      include ProfessorName[professor]
from TEACHINGS
      selected by offering = o

node ProfessorName [p : Professor]
      list_type: none
   is
      href Professor[self] name
from PROFESSORS
```

The `Course` node is the top of the nested structure, it contains data coming directly from Course objects (credits, description) and it includes nodes `Prereqs` and `Offerings` which contains the lists of prerequisites and offerings respectively. The node `Offerings` displays information about particular offerings for this course (code, semester). It includes a node Teaching which displays the list of professor names for this offering. Note that the "list_type" statement allows to specify different types of presentations (bullet lists, definition lists, enumerations with a separator character, etc.). Figure 3 shows a typical instance of `Course`.

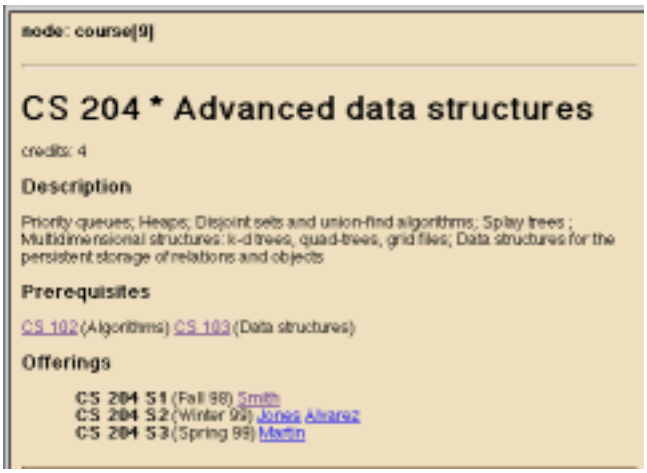

Figure 3. A node with a nested structure

Aggregation can also occur on objects which are not related in the database. For instance, a university home page may point to courses, research, and social events which are not related entities of the database.

3.5 Recursive Inclusions (Static and Dynamic)

The following node schema is recursive since it has an inclusion link to itself

```
node EmpWithMgr [e : Emp ] is
  " No: ", bold(no), " => " , bold(name),
  break(), " Hire date: ", hire_date,
  include EmpWithMgr [manager]
from Employees selected by self = e
  ordered by empno
```

However this does not generate a cyclic inclusion structure at the instance level since the graph of the *manager* relation is acyclic (i.e. there is no cycle at the data level). Fig. 4 shows an instance of that node schema.

Figure 4. Recursive inclusions of nodes

When there are cycles in the data, the visibility distance mechanism prevents the creation of recursive inclusions at the instance level. Since each field in a node definition has a maximum visibility distance, it implies that there is a level of inclusion from which all the included nodes become empty.

Thus, recursive inclusion can be employed to specify hypertext views that represent data having a tree or graph structure. This representation may significantly reduce the number of navigation steps, compared to the direct object mapping.

3.6 Representing Specialized Entities (Union Nodes)

Collections may contain objects of different classes which are subclasses of the collection's class. To represent such polymorphic sets one can use the inclusion mechanism. A top node is used to represent the common attributes, it has inclusion links to specific nodes used to represent the specific parts. Each specific node selects only the objects which have a given type.

```
// generic node
node emp [e: employee] is
  name, address, ... // common attributes
  include driver[self]
  include secretary[self]
from EMP

// specific nodes
node driver [e: employee] is
  max_km, license_no, ...
from EMP/Driver selected by self=e

node secretary[s: employee] is
  ...
from EMP/Secretary selected by self=s
```

An expression of the form *collection*/*class* represents all the objects in *collection* which are instances of *class* or instances of a subclass of *class*.

3.7 Previewing and Outlining Linked Nodes [10]

Previewing a node M from a node N consists in including in N part of the content of M and a reference link to M. The partial inclusion can be obtained with the visibility mechanism (some field must have visibility 0). The aim of previewing is to give information about the content of a node without having to navigate to it.

4 Formal Semantics of the Language

The definition of the semantics is quite similar to the specification of database query languages but it must also take into account the notion of node identity which is necessary to specify the semantics of links. The identity of a node instance is a triple (*schema name*; *actual parameter values*, *inclusion level*). This differs from the semantics of usual query languages which do not create new objects as the result of a query.

To formally define the semantics of the definition language we must specify, for a given database instance, how to interpret a node instance expression in terms of nodes, links, anchors and contents (the complete syntax of the language is given in appendix). To keep the description small, we will suppose that the semantics of arithmetic and logic expressions is given and we will not take into account presentation functions.

Let D be the node definition:

node N [p_1:T_1, p_2: T_2, …, p_k: T_k]
f_1, …, f_n
from C
selected by S(*self*, p_1, p_2, …, p_k)
ordered by O(*self*, p_1, p_2, …, p_k)

where each field f_j is

level l_j [(**href** | **include**) N'_j**[***expr'*$_1$, *expr'*$_2$, …, *expr'*$_{sj}$**]**] e_j

The interpretation of a node instance expression $E = N[expr_1, expr_2, \ldots, expr_k]$ at a given inclusion depth d is composed of a *node identity* $\mathbf{I_{id}}(E, d)$, a *node content* $\mathbf{I_C}(E, d)$, and a *set of links* $\mathbf{I_L}(E, d)$.

The identity of a node consists of the node's name, the value of its parameters and its inclusion depth.

$$\mathbf{I_{id}}(E, d) = (N, [\mathbf{I}(exp_1), \mathbf{I}(exp_2), \ldots, \mathbf{I}(exp_k)], d).$$

In order to define the content of the node, we first define the set of objects from which the content will be drawn:

$$S_0 = \{o \in C \mid \mathbf{I}(S)(o, \mathbf{I}(expr_1), \mathbf{I}(expr_2), \ldots, \mathbf{I}(expr_k)) = \text{true}\}.$$

It is the set of objects o that belong to C and satisfy the predicate S.

Then S_1 is a sequence $<o_1, o_2, \ldots, o_r>$ such that

$$o_i \in S_1 <=> o_i \in S_0 \text{ and}$$

$$\mathbf{I}(O)(o_i, \mathbf{I}(expr_1), \mathbf{I}(expr_2), \ldots, \mathbf{I}(expr_k)) \leq \mathbf{I}(O)(o_{i+1}, \mathbf{I}(expr_1), \mathbf{I}(expr_2), \ldots, \mathbf{I}(expr_k)).$$

Thus S_1 is the set S_0 ordered by the expression O.

The content $\mathbf{I_C}(E, d)$ of the node is the sequence of elements obtained by concatenating the sequences $<\mathbf{I_C}((f_1, o_i, d), \ldots, \mathbf{I_C}(f_n, o_i, d)>$ $(i = 1, r)$ where $\mathbf{I_C}(f_j, o_i, d)$ is the content of field f_j for object o_i at depth d. It is defined as follows:

- $\mathbf{I_C}$ (**level** l_j <link-specification> e_j, o_i, d) = <empty> if the visibility level l_j is less than d,
- $\mathbf{I_C}$ (**level** l_j <link-specification> k, o_i, d) = k if k is a constant,
- $\mathbf{I_C}$ (**level** l_j <link-specification> a, o_i, d) = $o_i.a.toString()$ if a is an attribute name (where *toString* is a method that maps an object to its string representation).

The set of links $\mathbf{I_L}(E, d)$ is the union of the sets $\{\mathbf{I_L}((f_1, o_i, d), \ldots, \mathbf{I_L}(f_n, o_i, d)>$ $(i = 1, r)$ where

- $\mathbf{I_L}$(**level** l_j e_j, o_i) = the null link
- $\mathbf{I_L}$(**level** l_j **href** $N'_j[expr'_1, expr'_2, \ldots, expr'_{sj}]$ e_j, o_i) is a reference link with
  starting node id: $\mathbf{I_{id}}(E, d)$ (the id of this node instance),
  ending node id: $\mathbf{I_{id}}(N'_j[expr'_1, expr'_2, \ldots, expr'_{sj}], d)$,
  starting anchor: $(i–1)n + j$ (the sequence number of this element),
- $\mathbf{I_L}$(**level** l_j **include** $N'_j[expr'_1, expr'_2, \ldots, expr'_{sj}]$ e_j, o_i) is an inclusion link with
  starting node id: $\mathbf{I_{id}}(E, d)$
  ending node id: $\mathbf{I_{id}}(N'_j[expr'_1, expr'_2, \ldots, expr'_{sj}], d+1)$
  starting location: $(i–1)n + j$

5 Comparison with Related Work

Database publishing. Several ways have been explored to publish the content of databases on the Web. The procedural approach consists in writing database programs that generate HTML pages (e.g. Oracle Web Server [14], CGI scripts, Java/JDBC server-side applications, etc.). Another approach consists in automatically generating HTML pages from the database schema (e.g. O2Web [23]).

This last approach corresponds to the direct object mapping described in section 3.1. It can be improved by defining a collection of views over the database and generating the hypertext from these views; it is also possible to overload generic methods with specific ones. However, it is not clear that features like (recursive) inclusion links can be easily expressed with this technique.

Toyama and Nagafugi [18] define and extension of the SQL query language to present the result of a query as a structured document (e.g. HTML, LaTeX). They introduce connectors and repeaters in place of the SQL target list of a query.

Virtual documents. The virtual document approach consists in extending a document definition language with database querying features. For instance, database queries can be embedded into HTML pages [13]. In [15] Paradis and Vercoustre propose a prescription language to specify the static and dynamic content of a virtual document. The static content is expressed with the usual HTML tags. The dynamic content is obtained by evaluating queries on (heterogeneous) data sources. The language has operators to select and combine information from different query results.

Web site management. In [8], Fernandez et al. describe a system to produce a Web-site (a set of HTML pages) from different data sources integrated through a graph based data model. A specific query language (STRUQL) is used to query the data graph and construct a graph that forms the content of the Web-site. A second language (HTML-template language) is used to specify the presentation of each object.

In [16] Siméon and Cluet extend the YAT system to build HTML pages. The YATL language allows to specify graph conversions between the input data model (ODMG objects, XML documents, …) and the output model (HTML pages) viewed as graphs.

Methodology for the Design of Web Applications. The Araneus methodology proposed by Atzeni et al. [2] distinguishes three levels: the hypertext conceptual level (Navigation Conceptual Model); the hypertext logical level (Araneus Data Model); and the presentation level (HTML templates). It is possible to analyze our language with respect to these levels. The conceptual level corresponds to node names, reference links, base collections, and selection predicates. The logical level (internal node structure) corresponds to the specification of node fields (constants, attributes and inclusion links). A node with a complex ADM type can be represented by a hierarchy of included nodes (see 3.4). Finally, the markup functions (or strings with HTML tags) define the generated document's markup which will be used to present the document.

In [9] Fraternali and Paolini introduce the HDM-lite methodology which is an adaptation of the Hypermedia Design Methodology for Web applications. Their navigation model includes navigation modes (index, guided tour, showall, …) to help navigating within collections of objects.

6 Conclusion

Language properties. The language we have presented has several properties that are important to develop hypertext views: it is non-procedural, it has the capacity to restructure the database information, i.e. to present it in different forms (corresponding to different points of view), it has the capacity to create structured node contents (complex hierarchical nodes or structured documents), it has the capacity to create orientation structures like indices, outlines, node previews, etc., it has a node identification scheme

that enables other applications to access the generated nodes and that permits to store hypertext views independently of the database.

A view definition is relatively robust with respect to schema updates since every node schema depends only on its base collection. In addition, a node schema depends only on the name and the parameter of the node it refers to. Thus node schemes can be changed without affecting the rest of the hypertext view.

From a more theoretical point of view, we have already shown in [7] that select-project-outer-join queries can be represented by nodes with inclusion links. To represent select-project-join queries it is necessary to slightly modify the semantics of inclusion links.

Prototypes. We have developed several tools to generate hypertext views. The LAZY system generates Web pages over a relational database, it implements a subset of the described view language. The implementation relies on two components: a node definition compiler and a node server connected to a HTTP server. The compiler translates node definitions into relational views and stored procedures. The node server dynamically generates HTML pages by querying the generated views and/or calling the stored procedures. We are currently working on a portable implementation of the node server, written in Java and based on JDBC.

The MetaLAZY tool is an implementation of the hypertext view language for a semantic data model. It translates a semantic data schema into a relational schema and generates LAZY nodes for this schema. It uses multiple levels of inclusion to hide the auxiliary relations that represent many-to-many relationships.

We have also developed a tool to produce materialized hypertext views over an O_2 database. These views consist in sets of HTML pages that can be stored on external media (CD-ROM, etc.) independently of the database.

Future plans include the addition of navigation modes [9] to node schemes and mechanisms to update the database through hypertext views.

Aknowledgement

We would like to thank the anonymous referees for their insightful and valuable comments.

7 References

1. S. Abitboul, R. Hull, V. Vianu. *Foundations of Databases*, Addison-Wesley, 1995.
2. P. Atzeni, G. Mecca, P. Merialdo. “Design and Maintenance of Data-Intensive Web Sites”.In Proc. of the EDBT’98 Conf., Valencia, 436-450, 1998
3. T. Barsalou, N. Simabela, A. Keller, G. Wiederhold. “Updating Relational Databases through Object-Based Views”. In Proc. ACM SIGMOD, Denver, 248-257, 1991.

4. T. Berners-Lee, R. Cailliau, A. Luotonen, H. Frystyk Nielsen, A. Secret. “The World-Wide Web”. Comm. of the ACM, Vol. 37, No. 8, 76-82, 1994..

5. S. Bobrowski. *Oracle7 Server Concepts Manual*, Oracle Corp., Redwood City, CA, 1992.

6. J. Conklin. ”Hypertext: An Introduction and Survey”. IEEE Computer, Vol. 20, No. 9, 17-42, 1987.

7. G. Falquet, L. Nerima, J. Guyot. “A Hypertext View Specification Language and its Properties”. CUI Technical report #102, University of Geneva, 1996.

8. M. Fernandez, D. Florescu, J. Kang, A. Levy, D. Suciu. “Catching the Boat with Strudel: Experiences with a Web-Site Management System”. In Proc. ACM SIGMOD Conf., Seattle, 414-425, 1998.

9. P. Fraternali, P. Paolini. “A Conceptual Model and a Tool Environment for Developping More Scalable, Dynamic, and Customizable Web Applications”. In Proc. of the EDBT’98 Conf., Valencia, 421-435, 1998.

10. S. Ichimura, Y. Matsushita. "Another Dimension to Hypermedia Access". In Proc. of the Hypertext'93 Conf., Seattle, 63-72, 1993.

11. C. Lécluse, P. Richard, F. Velez. "O2, an Object-Oriented Data Model ". In Proc. ACM SIGMOD, Chicago, 1988.

12. J. Nanard, M. Nanard. "Shoud Anchors Be Typed Too? An Experiment with MacWeb". In Proc. of the Hypertext'93 Conf., Seattle, 51-62, 1993

13. T. Nguyen, V. Srinivasan. “Accessig Relation al Databases from the World Wide Web”. In Proc. ACM SIGMOD Conf., 529-540, 1996.

14. Oracle Inc. home page: http://www.oracle.com

15. F. Paradis, A-M. Vercoustre. “A Language for Publishing Virtual Documents in the Web”. In Proc. of the WebDB Workshop, Valencia, 1998.

16. J. Siméon, S. Cluet. “Using YAT to Build a Web Server”. In Proc. of the WebDB Workshop, Valencia, 1998.

17. J. Teuhola. “Tabular Views on Object Databases”. Tech. Rep. R-93-11, University of Turku, Finland, 1993

18. M. Toyama, T. Nagafuji. “Dynamic and Structured Presentation of Database Contents on the Web”, In Proc. of the EDBT’98 Conf., Valencia, 451-465, 1998.

19. C. A. Varela, C. C. Hayes. “Zelig: Schema–Based Generation of Soft WWW Database”. In Proc. W3 Conf., 1994.

20. *Special Issue: Advanced User Interfaces for Database Systems*. SIGMOD Record, Vol. 21, No. 1, 1992.

21. Special section: Hypermedia Design, CACM, Vol. 38, No. 8, 1995.

22. “The O2 System”. Comm. of the ACM, Vol. 34, No. 10, 1991

23. “O2 Web Presentation”, O2Technology, Versailles, France, 1995.

Appendix

Syntax of the LAZY language

HypertextView = **define** { Node-schema } **end**

Node-schema = **node** node-name ["[" Parameter-list "]"]
List-Markup
is
Field-list
from collection ["/" type-name]
[**selected by** Expression]
[**ordered by** Expression-list]

Field-list = Field { "," Field }

Field = [Level] [Link-spec] Markup-Element

Markup-element = Element | Markup "(" Markup-element ")"

Element = ε | Constant | attribute-name

Link-spec = (**href** | **include**) node-name ["[" Expression-list "]"

Parameter-list = Parameter { "," Parameter }

Parameter = [**set**] param-name ":" type-name

Expression-list = Expression { "," Expression }

Expression = Term | Term Op Term

Term = Atom | "(" Expression ")"

Atom = Constant | attribute-name | **self**

Op = "=" | "<" | ">" | "+" | "-" | **and** | **or** | ...

Constant = string | number

Markup = **bold** | **italic** | **break** | ... | **heading1** | **heading2** | …

List-markup = list_type: (**ordered** | **unordered** | **definition** | **enumeration | none**) [(separator)]

Level = **level** "0" | ... | "9"