Querying Structured Documents with Hypertext Links using OODBMS*

V. Christophides
I.N.R.I.A.
Domaine de Voluceau, Rocquencourt B.P. 105
78153 Le Chesnay, Cedex France
e-mail: Vassilis.Christophides@inria.fr

A. Rizk
Euroclid
12 Avenue des Prés
78180 Montigny le Bretonneux France
e-mail: Antoine.Rizk@inria.fr

ABSTRACT

Hierarchical logical structure and hypertext links are complementary and can be combined to build more powerful document management systems [28, 25, 24, 13]. Previous work exploits this complementarity for building better document processors, browsers and editing tools, but not for building sophisticated querying mechanisms. Querying in hypertext has been a requirement since [19] and has already been elaborated in many hypertext systems [11, 7, 4, 21], but has not yet been used for hypertext systems superimposed on an underlying hierarchical logical structure.
In this paper we use the model and the SQL-like query language of [10] in order to manage structured documents with hypertext links. The model represents a structured document with typed links as a complex object, and uses paths through the document structure, as first class citizens in formulating queries. Several examples of queries illustrate, from a practical point of view, the expressive power of the language to retrieve documents, even without exact knowledge of their structure in a simple and homogeneous fashion. It must be stressed that the proposed model and language implement the equivalent HyTime [1] Location Address Module. In fact, the language is more powerful than the corresponding HyQ query facilities. The implementation and the description throughout the paper use the SGML standard [2] to represent the document structure and the object-oriented DBMS O_2 [12] to implement the query language and the storage module.
KEYWORDS: structured documents, hypertexts, object oriented databases, information retrieval, query languages, path expressions.

*Work partially supported by Esprit Project *TecPad*.

1 INTRODUCTION

The combination of hierarchical structure with hypertext links is not a new idea. On the one hand, early hypertext systems [18, 3] proposed the idea of hierarchy as a means of imposing structure on an otherwise unordered graph of information. This kind of hierarchy, that most often takes the form of complex nodes, proved to be a helpful feature in browsing large hypertexts as well as for decomposing large graphs into submodules (for printing, collaborative work etc.), but not for building sophisticated querying mechanisms. Until recently [15, 23] this hierarchy in hypertext systems, remained instance oriented, i.e., it is not defined at a generic level that could be used as a type for further instantiation. On the other hand, structured document systems, which make use of generic logical structures, do implement hypertext features, but these features are just cross-references and simple annotations.

In [28], it has been shown that hypertext features and document logical structure are indeed complementary and can be combined in a very powerful manner. In this work, the hypertext links point to subtrees in the document instance and use paths from the tree root to identify subtrees, as well as tree pattern matching for more powerful intrinsic linking. SGML [2] allows the representation of untyped hypertext links between document elements through the use of the ID/IDREF feature. In the Grif document system [24], links were further typed according to the generic logical structure of which the target subtree is an instance. The creation of a link has therefore to correspond to the document type definition (DTD). Its scope could be constrained to the DTD or to a set of DTDs, and its physical presentation could be generated automatically in conformance with a generic physical structure.

Recently, many researchers have studied the problem of hypertext querying (GraphLog [11], Beeri [7], GOOD [17], GRAM [4] and MORE [21]). These approaches view hypertext as a graph of information chunks which do not have an underlying backbone of logical hierarchical structure. On the other hand, much work has been devoted to querying structured -mainly SGML- document bases, but without taking seriously into account the hypertext links involved (see Maestro [22], Textriever [9] and SGML-Search [8], DynaText[13]). We believe that for *very large text databases* where the interconnected information does not have a flat structure, the combination of navigation through the documents structure (e.g., tree, graph) with document retrieval by content, is very rewarding. At this point we must note that recent evolution of document standards, such as HyTime [1], proves the importance of combining a logical hierarchical structure with hypertext features for documents querying and navigation.

In [10] a model and a query language that are specifically oriented to managing structured documents have been proposed. In this paper we show that this model is also adequate for exploring structured documents with typed links via an SQL-like query language. The model represents a structured document with hypertext links as a complex object, where the paths through the object structure are used, as first class citizens, in formulating queries. Compared to other approaches [29, 26] paths (including hypertext links) are defined in conformance with the generic logical structure of a document (DTD) and do not require additional data structure for their creation and storage. Several examples of queries illustrate, from a practical point of view, the expressive power of the language to retrieve documents, even without an exact knowledge of their structure, in a simple and homogeneous fashion. Finally, the expressive power of the proposed SQL-like language with respect to HyTime queries is also illustrated : examples of queries are given in both the described language and the HyQ query language [20].

The implementation uses SGML [2] and the OODBMS O_2 [12] throughout. In particular, an extension of the Euroclid SGML parser [14] has been developed to translate SGML documents into O_2 schemas and instances. Thus, an SGML DTD is automatically translated into an O_2 schema and document instances are also decomposed into O_2 objects and values in conformance with the O_2 schema. The described query language is currently being developed on top of O_2SQL [6].

The paper is organized as follows : Section 2 provides a very brief overview of basic SGML. Section 3 describes the translation from SGML to O_2 both at the generic and instance levels. Section 4 presents the main part of our work namely the query language with examples of querying document contents, hierarchical structure and hypertext links. Section 5 provides a comparison of our work with the HyTime standard and its query language HyQ. Finally, Section 6 concludes the paper.

2 BASIC SGML

```
 1. <!DOCTYPE   article [
 2. <!ELEMENT   article    - -    (title, author+, affil, abstract, section+, summary?)>
 3. <!ATTLIST   article           status  (final | draft)  draft>
 4. <!ELEMENT   title      - -    (#PCDATA)>
 5. <!ELEMENT   author     - O    (#PCDATA)>
 6. <!ELEMENT   affil      - O    (#PCDATA)>
 7. <!ELEMENT   abstract   - O    (#PCDATA)>
 8. <!ELEMENT   section    - O    ((title, content+) |  (title, content*, subsectn+))>
 9. <!ELEMENT   subsectn   - O    (title, content+)>
10. <!ATTLIST   subsectn          label  ID  #IMPLIED>
11. <!ELEMENT   content    - O    (figure | paragr)>
12. <!ELEMENT   figure     - O    EMPTY>
13. <!ATTLIST   figure            label  ID  #IMPLIED
                                  sizex   NMTOKEN  #REQUIRED
                                  sizey   NMTOKEN  #IMPLIED
                                  file    ENTITY #IMPLIED>
14. <!ENTITY      fig1    SYSTEM  "/u/christop/SGML/image1"   NDATA bitmap>
15. <!NOTATION bitmap  PUBLIC  "-//local//NOTATION Bitmap images//EN">
16. <!ENTITY      fig2    SYSTEM  "/u/christop/SGML/image2"   NDATA gif>
17. <!NOTATION gif       PUBLIC  "-//local//NOTATION Gif images//EN">
18. <!ELEMENT   paragr     - O    (#PCDATA)>
19. <!ATTLIST   paragr            reflabel  IDREF  #REQUIRED>
20. <!ELEMENT   summary - O       (#PCDATA)>        ]>
```

Figure 1: A DTD for a document of type article

In this section, we present the main features of basic SGML that are necessary for our work (for a general presentation of SGML see [16]). In order to define a document's logical structure, SGML adds descriptive markup (tags) in document instances. SGML tags serve two purposes : a) they identify each component of the logical structure, called elements, in the document instance, b) they show the syntactic relationships among these elements in the Document Type Definition (see Figure 1).

Each *element* has a name, a structure and some indications (e.g., "-O" indicates that the tag can be omitted if there is no ambiguity). For example, line 2 of Figure 1 defines the structure of the element with name article. The element structure is built using other elements or basic types such as #PCDATA, EMPTY, etc. The former basic type represents strings possibly combined with other elements that need further parsing, while the latter represents elements that do not have any content (empty space). Basic types and elements are combined using connectors and elements can be further qualified with occurrence indicators. In particular the following can be used :

- The aggregation connector (",") implies an order between elements. For example, a subsection is composed of a title followed by contents (line 9).
- The choice connector ("|") provides alternative structures for the same element. For instance, the element content is either a figure or a paragraph (line 11).
- The optional indicator ("?") indicates zero or one occurrence of an element. For instance, an article may or may not have a summary (line 2).

- The plus sign ("+") indicates one or more occurrences of an element (e.g., section+ in articles, line 2); and the asterisk ("*") zero or more occurrences (e.g., content* in element section, line 8).

A finite number of *attributes* can be associated with each element adding further information (i.e. semantics). Each attribute has a name, a declared domain and, possibly, a default value. A domain can be either an SGML type (e.g., in line 13 the type of the attribute sizex is NMTOKEN) or a range (e.g., in line 3 the attribute status ranges over final and draft and has draft as a default value). For cross references between elements, one must use attributes whose domains are ID (for the element that will be referenced) and IDREF (for the element referencing it). For instance, paragraphs can have references to subsections or figures (Figure 1 lines 19, 10 and 13). It must be stressed that in SGML this kind of hypertext links are not typed. The HyTime standard [1] builds on this basic feature and develops a very sophisticated mechanism for addressing elements according to their names, attributes, logical structure or according to their localization on a multidimensional coordinate system. In this way, HyTime extends the traditional SGML addressing mechanism beyond the scope of a single document and to elements where the feature (ID/IDREF) has not been defined (see Section 5).

Finally, *entities* represent units of information storage that contain part of a document (e.g., images, subdocuments, etc.) with a content type possibly outside of the SGML reference syntax. An entity can be referenced by its name, from one or more places in a document instance or a DTD. There are two kinds of entity references. First, references to named constants or macros (i.e. general and parameter entities) and second, references to external storage objects. In the example, an external entity is used to define the attribute file (line 13) whose value will be found in a file (fig1 or fig2, lines 14 and 16) of another system (keyword SYSTEM) and should not be parsed by SGML (type NDATA). The *notations* associated with these entities (bitmap and gif, lines 15 and 17) declare the corresponding format of the referenced data.

3 MAPPING SGML TO O_2

The problem of mapping an SGML DTD into an O_2 [12] schema and a document instance into corresponding objects and values is studied in [10]. In this section we extend this approach to represent typed hypertext links and different data notations associated with external entities. We must note that the DTDs considered in this paper support basic SGML (see Section 2) without exceptions (e.g., inclusions and exclusions).

As SGML documents are always hierarchical structures of elements, they can be naturally represented using the O_2 hybrid data model including complex values/objects[1]. The tuple and list constructors are very useful to describe the structure of SGML elements. Cross references between elements can be easily modeled using object identity. Furthermore, object sharing provided by OODBMSs is useful for the manipulation of common document components (e.g., external entities). However, SGML document modeling leads to other requirements mainly related to the use of connectors and occurrence indicators mentioned in Section 2. None of the existing OODBMS provides all the features we need. We chose O_2 because of (i) its sophisticated type system and (ii) its query language that can easily be extended. In the following, we assume the existence of the following features:

Ordered tuples: The ordering of elements is meaningful in SGML. Sometimes, the ordering is imposed (","); sometimes some flexibility is left ("&"). In [10] a solution has been proposed based on polymorphism that introduces ordered tuples in the O_2 data model with respect to the other type constructors such as lists. More precisely, another way of viewing ordered tuples is as heterogeneous lists.

Union of types: In [10] there also has been added the type union in order to provide alternative structures for the same SGML element type ("|"). We must note that typing implies that the components of a union type are *marked*, i.e., a tag (attribute) is used in order to specify the alternative type that has been chosen in an element instance. In the sequel we simply refer to union types.

Each SGML element definition is now interpreted as a class having a type, some constraints and a default behavior (i.e., standard display, read and write methods for each attribute). For instance, Figure 2 presents the O_2 classes corresponding to the elements definition in the DTD of Figure 1.

An SGML basic type is represented by an O_2 class of an appropriate type (e.g., `Text`). The aggregation connector (",") is modeled by ordered tuples (e.g., `class` Article) and the choice connector ("|") by a union type (e.g., `class` Content). Tuple (or union) attributes are named with the element's names (e.g., tags) in lower case letters (e.g., title, abstract, etc.). For unnamed SGML elements defined through nested parentheses (e.g., (title, content+) line 8, Figure 1), system supplied names are provided (e.g., a_1 in class Section, Figure 2). Elements without connectors imply an inheritance link with the corresponding O_2 class (e.g., `class` Title). Note that SGML attributes enter into the O_2 class type definition (e.g., status in `class` Article) as private attributes (i.e., not displayed). Note also that in contrast

[1] In this section, we assume standard knowledge of object-oriented DBMS (OODBMS).

```
class Article    public type tuple (title: Title, authors: list (Author), affil: Affil,
                                    abstract: Abstract, sections: list (Section),
                                    summary: Summary, private status: string)
                 constraint:        title != nil, authors != list(), abstract != nil,
                                    sections != list(), status in set ("final", "draft")
class Title      inherit            Text
class Author     inherit            Text
class Affil      inherit            Text
class Abstract   inherit            Text
class Section    public type union  (a1: tuple (title: Title, contents: list (Content)),
                                    a2: tuple (title: Title, contents: list (Content),
                                               subsectns: list (Subsectn)))
                 constraint:        (a1.title != nil, a1.contents != list())
                                    | (a2.title != nil, a2.subsectns != list())
class Subsectn   public type tuple  (title: Title, contents: list (Content),
                                    private label: string)
                 constraint:        title != nil, contents != list()
class Content    public type union  (figure: Figure, paragr: Paragr)
                 constraint:        figure != nil | paragr != nil
class Figure     public type tuple  (private label: string, private sizex: string,
                                    private sizey: string, private file: Entity)
                 constraint:        (sizex != nil)
class Paragr     inherit            Text
                 public type tuple  (private reflabel: union (subsectn: Subsectn,
                                                              figure: Figure))
                 constraint:        (reflabel.subsectn != nil | reflabel.figure != nil)
class Summary    inherit            Text
name Articles:   list (Article)
class Entity     public type tuple (private type: string, private format: string,
                                    private source: string)
name fig1:       Entity
name fig2:       Entity
```

Figure 2: O_2 classes for documents of type article

to SGML, cross references are now typed by their extremities (e.g., attribute reflabel in `class` Paragr). The elements marked by a "+" or "*" occurrence indicator are represented by lists (e.g., attribute authors in `class` Article). Finally, named values are used as persistent roots of documents (e.g., `name` Articles).

The constraints associated to a class are derived from occurrence indicators and from attributes defined with the keyword #REQUIRED or with range domains. The absence of an occurrence indicator on a component implies that the corresponding O_2 element is different from `nil` (e.g., attribute title in `class` Article). The "+" occurrence indicator is translated by a constraint specifying that the corresponding list must be different from the empty list (e.g., attribute authors in `class` Article). Attributes defined with the keyword #REQUIRED must be different from `nil` (e.g., sizex in `class` Figure). Range domains are translated by `in` constraints (e.g., attribute status in `class` Article).

Since SGML entities (e.g., fig1, line 14, Figure 1) are independent of elements definitions they are represented by named objects of a specific `class` Entity. To some extent we can consider the structure of an entity as "virtual" because the mapping from entities to real storage of data is implementation-dependent, and there is not a one-to-one relationship between entities and storage objects. Thus, `class` Entity can be viewed as a library of external data sources. In this fashion each entity (e.g., fig1) has a type (e.g., NDATA), possibly a format related to the interpretation of its data (e.g., bitmap) and a reference to the storage object on a physical system or in a single file (e.g., "/u/christop/SGML/image1"). References to entities can be implemented using object identity (e.g., attribute file in `class` Figure). Finally, it must be stressed that multimedia types can easily be incorporated into O_2 (e.g., bitmap and gif images). A redefinition of the system provided `method` display() in `class` Entity makes it possible to display, on demand, the related data stored outside of the object-oriented database, using the appropriate format.

To conclude this section, it should be noted that modeling references between document elements using object composition, misuses the original semantics of hypertext links. Indeed, a link can be typed not only by its extremities, but also explicitly by its information content [11, 4]. Thus, links must be represented as independent objects of a specific type and not as properties of its extremities elements.

4 THE EXTENDED O_2SQL

In the SGML world, structured documents are usually queried by means of Information Retrieval Systems (IRS) which provide two facilities lacking in standard OODBMS query languages. IRSs [22, 9, 8, 13] provide sophisticated pattern matching facilities whose implementation relies on full text indexing techniques and do not require users to know the exact structure of the documents they query. However, in most cases, IRS query facilities are not integrated with navigation mechanisms even through cross references between elements into the document structure. On the other hand, in hypertext systems, if a query mechanism is provided, it mainly concerns retrieval by structure through the documents graph and does not take into account the underling logical structure of the information chunks [11, 7, 17, 4, 21]. We believe that for *very large text databases* where the interconnected information does not have a flat structure, the combination of navigation through the documents structure (e.g., tree, graph) with document retrieval by content, is very rewarding.

In this section, we use the extended O_2SQL query language proposed in [10], in order to query structured documents with hypertext links. This language (i) provides sophisticated string predicates in queries, (ii) uses paths through the objects structure as first class citizens (iii) takes into account union of types and ordered tuples[2]. These features lead to a language that smoothly combines the features of both IRS and database access languages with navigation facilities. The presentation of the extended O_2SQL relies on examples. The running example uses the O_2 schema of Figure 2.

4.1 Querying Structured Documents

In order to query data with incomplete knowledge of its structure a new sort, PATH, has been introduced in [10]. A value of this sort denotes a *concrete path* through complex objects/values (crossing objects, tuples, lists, etc.). Intuitively, concrete paths allow navigation through document's specific structures. For instance, Figure 3 illustrates the specific structure of an article complex object defined in Figure 2. Thus, $my\_article.sections[1].subsectns[0].title$ is a concrete path. This path selects the attribute *sections* of *my_article*, then the second section, the attribute *subsectns* of this section and finally, the title of the first subsection.

With this approach *paths can be queried like any other data.* Like all types manipulated in O_2SQL, PATH comes with its variables and basic queries. To distinguish variables according to their sort (standard data or PATH), path variables are prefixed by "*PATH_*". More specifically *concrete paths* are sequences of:

1. "$\cdot a$" where a is an attribute name (for tuples or marked unions);
2. $[i]$ where i is an integer (for lists);
3. $\rightarrow$ (for objects dereferencing[3]);
4. $\{v\}$ where v is a value (for sets).

A calculus for path querying and an algebraization of an SQL-like language have been proposed in [10]. In this paper we are only interested in the use of path expressions in the extended O_2SQL. The query below uses standard path expressions (i.e., without path variables) such as are provided by O_2SQL, and introduces a textual predicate (`contains`).

Q1: *Find the title and the abstract of articles having a section with a title containing the words "SGML" and "OODBMS".*

```
select tuple (title: a.title, abstract: a.abstract)
from   a in Articles, s in a.sections
where  s.title contains ("SGML" and "OODBMS")
```

The `contains` predicate allows a string (e.g., a word or a sentence) to be matched with a pattern or a boolean combination of patterns using concatenation, disjunction, Kleene closure, etc. Two words in a sentence may be separated by blank characters or by new line indicators ignored by the users. Thus, at the language level, sentences are just considered as a sequence of words or word patterns and it is up to the system to ignore non significant characters (e.g., separators) in the queried instances. Among other textual predicates, we only cite `near` that makes it possible to check whether two words are separated by, at most, a given number (n) of characters (or words) in a sentence. Finally, other content predicates associated with different data notations (see section 2) can be implemented by appropriate user defined methods but they are not used in an optimized way during query evaluation.

[2] Integrity constraints can be used for optimizing queries, they do not require new functionalities should to be added to O_2SQL.

[3] The symbol "$\rightarrow$" is used for crossing objects while "." is used for values. Since O_2SQL violates objects encapsulation, the two symbols can used in queries without any difference.

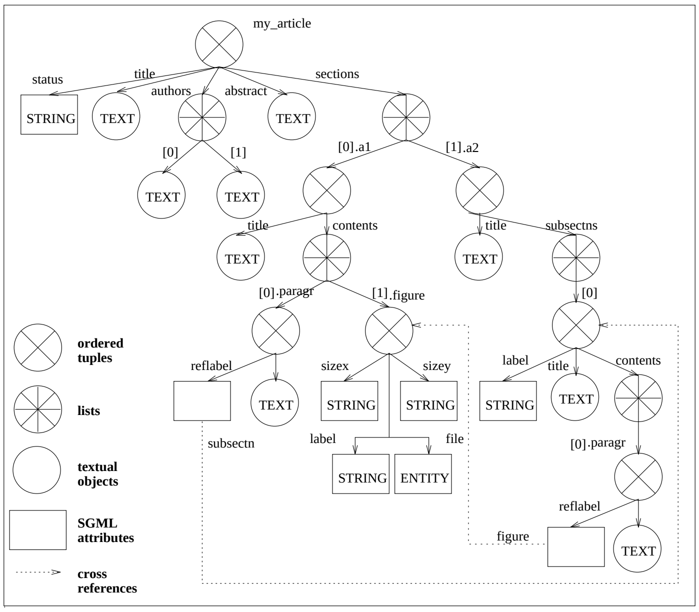


Figure 3: A complex object of type article

The following examples will illustrate the use of path variables in query formulation. Consider the query :

Q2: *Find all titles in my_article.*
`select` t
`from` $my\_articlePATH\_p.title(t)$

The result is a set of titles reached from the root of *my_article*, by following various paths in the article structure of Figure 3 (general title, section titles, etc.).

In query Q2, the subquery $my\_articlePATH\_p.title(t)$ illustrates a new basic query. It is a path expression with variables (here $PATH\_p$ and t), whose semantics differs from that traditionally used in path expressions without variables (see Q1). These queries return a set of tuples with one attribute per variable. In the example, it returns a set of tuples with two attributes: t, $PATH\_p$. The attribute $PATH\_p$ ranges over all concrete paths that start from the root of *my_article* and that end with an attribute named *title*. The value of attribute t in tuple $(t, PATH\_p)$ corresponds to the title that can be reached from *my_article* following the corresponding concrete path, value of the attribute $PATH\_p$. It must be stressed that if we are not interested in the actual values of the path variable we may allow the syntactical sugared form : `from` *my_article..title*(t)

Paths is a data type that comes equipped with functions. In particular, list functions can be used on paths. For instance, suppose that P is a concrete path of value *.sections[0].contents[0]* we can compute the length of P: `length`$(P) = 2$ and project P on its two first elements: $P[0] = .sections[0]$. Consider now the query :

Q3: *Find the children and grandchildren elements of my_article.*
`select` x
`from` $my\_articlePATH\_p(x)$
`where` `length`$(PATH\_p)$ <= 2

The result of the subquery $my\_articlePATH\_p(x)$ in Q3 is a set of concrete paths reached from the root of *my_article*, two steps down in the article structure. The variable x returns in the `select` clause the corresponding elements (see Figure 3). If there does not exist an evaluation of a path variable with a concrete path from the root of persistence the `length` function returns zero. For instance, consider the query :

Q4: *Find the leaves of my_article.*
`select` x
`from` $my\_articlePATH\_p(x)$, $xPATH\_q$
`where` `length`$(PATH\_q) = 0$

In Q4, the subquery $xPATH\_q$ returns the set of concrete paths reached from an element x in the structure of *my_article*. The condition in the `where` clause guarantees that x is a leaf of *my_article*. The following queries will illustrate the power of the extended O_2SQL to manipulate heterogeneous lists and ordered tuples.

Q5: *Find the paragraphs of my_article just before a figure.*
`select` x
`from` $my\_articlePATH\_p[i].paragr(x)$,
$my\_articlePATH\_p[j].figure$
`where` j = i-1

The `from` clause of the query Q5 defines four variables : $PATH\_p$ that ranges over all concrete paths that start from the root of *my_article* and that end with a heterogeneous list of paragraphs and figures, i and j that range respectively over the positions of the corresponding list elements with attributes *paragr* or *figure* and x that ranges over paragraphs. The `where` clause is used to select the correct quadruples $(PATH\_p, i, j, x)$ and the `select` clause returns the corresponding paragraphs (x). It becomes clear that Q5 makes it possible to select the youngest elder siblings of paragraphs in the structure of *my_article* that are figures. Given the fact that ordered tuples are treated in [10], as a special kind of heterogeneous list, the same mechanism can be also applied to select the siblings of an attribute within an ordered tuple. Consider now a more intricate query :

Q6: *Find the last paragraph of the first section of my_article.*
`last` (`select` p
`from` $my\_article.sections[0]\ PATH\_p.paragr(p)$)

In Q6 the facts that *my_article* is an ordered tuple (see Figure 2) and sections, subsections and bodies are lists, conserve the natural ordering of paragraphs within the article. Thus, the result of the `select-from-where` filter is a list of paragraphs within the first section of *my_article*. The O_2SQL provided function `last` selects the corresponding last paragraph.

4.2 Navigating through Hyperlinks

The following examples illustrate the use of SGML attributes (ID/IDREF, see section 2) for representing hypertext links. It must be stressed that in the model presented in Section 3, these hyperlinks become typed according to the generic logical structure of which the target subtree is an instance. Note also that the only distinction between SGML attributes and the other components of the document (e.g., elements), is the fact that the former are represented by private attributes. Since O_2SQL violates objects encapsulation, SGML attributes can be retrieved using ad-hoc queries.

Q7: *Find the figures referenced within a section of an article by a paragraph containing the sentence "complex object".*

`select` $c.reflabel.figure$
`from` a `in` Articles, s `in` a.sections, c `in` s.contents
`where` c `contains` ("complex object")

Recall now that, in the schema of Figure 2, sections have a union type: a section marked with a_1 corresponds to a tuple with attributes *title* and *contents* and a section marked with a_2, to a tuple with attributes *title*, *contents* and *subsectns*. In addition, contents also have a union type : a content may be marked either with an attribute *figure* or *paragr*. In order to omit markers of union types in the `from` clause of Q7, (e.g., s.a_1.contents or s.a_2.contents) and avoid failure at evaluation of the condition in the `where` clause (e.g., to contents that do not have a paragraph), in [10] there has been introduced the notion of *implicit selectors*. Any operation on a variable ranging over a domain with union type implies an implicit selection. In Q7, both types of sections have an attribute *contents*. Furthermore, O_2SQL can use the subquery c `contains` ("complex object") in the `where` clause, to restrict c to the type `text` (e.g., to paragraphs). Finally, since in our schema paragraphs may have references to subsections or figures the subquery $c.reflabel.figure$ selects explicitly only the relevant figures.

We must note that the subquery $c.reflabel$ in the `select` clause makes possible to navigate through cross references between elements in the document structure (see Section 2). In the model presented in Section 3 we did not make any distinction between composition links in the generic logical structure of a document (i.e., a tree) and hypertext links between elements of the document or other documents (i.e., a graph). Thus, users can navigate, via a declarative query language, though complex document structures, in a quite natural way. The following queries illustrate the use of path expressions for querying hypertext references in a more intricate way.

Q8: *Find the elements that have a reference to the second subsection of the first section of my_article.*

`select` x
`from` $my\_articlePATH\_p(x)$
`where` $x.reflabel = my\_article.sections[0].subsectns[1]$

The subquery $my\_articlePATH\_p(x)$ in the from clause of Q8, selects all the elements of *my_article*. But the subquery $x.reflabel$ in the `where` clause is used to restrict the variable x only to elements of *my_article* where the attribute $reflabel$ is defined (e.g., to paragraphs) and they have a reference to a subsection. Finally, the subquery $my\_article.sections[0].subsectns[1]$ is used to select implicitly the second subsection of the first section of *my_article* (see Figure 3). The following query illustrates the navigation through the document structure without a precise knowledge of the hypertext links involved :

Q9: *Find the elements that have a reference to an element within the first section of my_article.*

`select` x
`from` $my\_articlePATH\_p(x)$,
$my\_article.sections[0]PATH\_q(y)$
`where` $x.reflabel = y$

In this case, Q9 returns the paragraphs that have a reference to a subsection or a figure in the first section of *my_article* (see Figure 3). To see another example of the power of path variables for querying data with incomplete knowledge of their structure, let's now consider the query :

Q10: *Find the paths to (reach) the elements referenced by an element within the first section of my_article.*

`select` `path_to_string`$(PATH\_q)$
`from` $my\_article.sections[0]PATH\_p.reflabel(x)$,
$my\_articlePATH\_q(x)$

The subquery *my_article.sections[0] PATH_p.reflabel(x)* in the `from` clause of Q10 selects the referenced elements within the first section of *my_article*. The subquery $my\_articlePATH\_q(x)$ retrieves the corresponding paths. Finally, `path_to_string` is a function defined on the sort PATH that returns a string defining a path (i.e., a sequence of attribute names, lists, etc.).

When path variables are used in a path expression, there is always the possibility of cycles (in the schema and in the data). For instance, consider the subquery $my\_article.sections[1]PATH\_p$. Assuming that all sections of *my_article* have subsections and paragraphs can have references to the subsections including them, possible valuations of $PATH\_p$ are (see Figure 3) : $\rightarrow subsetcns[0] \rightarrow contents[0].paragr \rightarrow$, $\rightarrow subsetcns[0] \rightarrow contents[0].paragr \rightarrow reflabel.subsectn \rightarrow contents[0].paragr \rightarrow$, etc. There are alternatives for interpreting such path variables :

A liberal semantics: One can allow paths that do not visit the same object twice (e.g., $subsectns[0]$). This forces the system to consider paths of unbounded length, i.e., length determined by the data and not the schema and to introduce a loop detection mechanism.

A restricted semantics: Path variables are alternatively interpreted by concrete paths with no two dereferencings of objects in the same class (vs. the same object). For instance, the path: $\rightarrow subsetcns[0] \rightarrow contents[0].paragr \rightarrow reflabel.subsectn \rightarrow contents[0].paragr \rightarrow$ will not be considered since it would involve two dereferencings of the `class` Subsectn.

In hypertext applications, navigation is crucial and the liberal semantics should be used. This requires including in the query language some form of transitive closure operator (see [4]). We believe that such a form of recursive navigation within the data structure is not necessary for structured documents. Indeed, in this context references of paragraphs to the subsections containing the paragraphs are not meaningful. Thus, we use the restricted path semantics. This guarantees safety of queries and the resulting language can be implemented with efficient algebraic techniques [10]. Observe also that queries going more deeply in the search can still be specified using paths of the form $P \rightarrow P'$, etc. For instance, consider the query :

Q11: *Find the elements within the first section of my_article referenced indirectly (at least two hops) by an element of the last section.*

`select` y
`from` `last` $(my\_article.sections)\ PATH\_p.reflabel(x)$,
$xPATH\_q.reflabel(y)$
`where` $my\_article.sections[0]PATH\_r(y)$

In the `from` clause of Q11 the subquery `last` (*my_article.sections*) returns the last section of *my_article* while *PATH_p.reflabel(x)* returns the elements directly referenced. The subquery x *PATH_q.reflabel(y)* returns an element reached from x. Finally, in the `where` clause the subquery *my_article.sections[0] PATH_r(y)* selects only the referenced elements (y) which are contained in the first section of *my_article*.

5 THE HyTime STANDARD

In this section we provide a basic overview of the HyTime ISO standard [1], in order to identify those elements that are relevant to our work. HyTime defines a powerful notation for representing structured documents that contain both time dependent elements and hyperlinks. The two aspects are elegantly viewed as applications of the same basic function, namely addressing. Addressing is therefore fundamental to HyTime. We can identify two basic addressing mechanisms, namely i) addressing an element encapsulated within an event scheduled on a coordinate system, and ii) addressing an element within a hierarchical (tree) logical structure. HyTime is structured into five parts called modules :

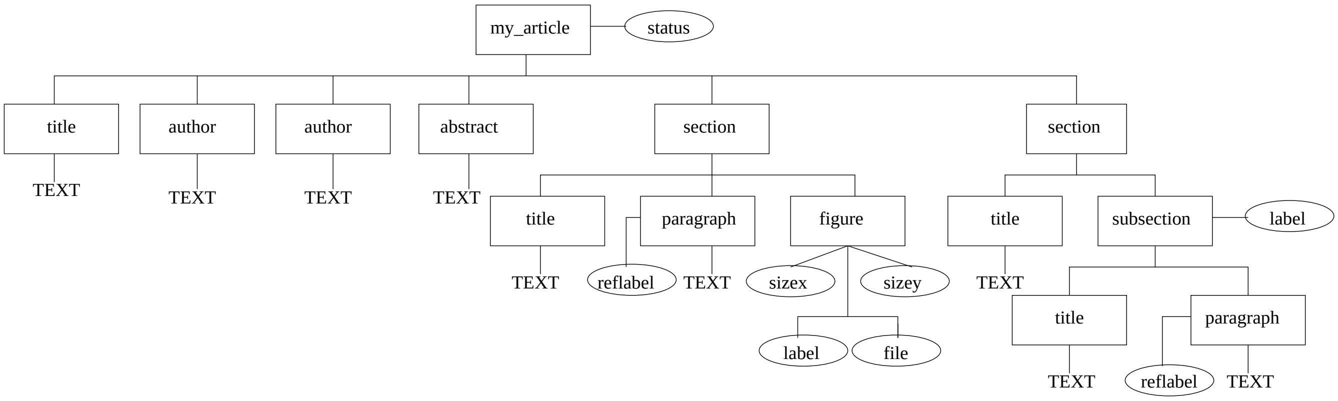
- The Base module contains the HyTime header declarations and general features.
- The Measurement module provides coordinate addressing capabilities, i.e., the basic functionalities for addressing ranges on a coordinate axis.
- The Location address module locates or address an element. Essentially, it allows one to identify SGML elements in many ways : by name, by unique ID, by position/path in a list/tree, or by coordinates on a Finite Coordinate Space when the Scheduling module is supported.
- The Hypertext module provides means of specifying a hypertext link as a couple of addresses, representing the source and the destination of the link.
- The Scheduling module allows one to create Finite Coordinate Space (FCS) systems, to encapsulate elements within events and to give to these events coordinates on the FCS. An event is an abstract notion that represents an encapsulating envelope containing “real” elements (e.g., multimedia information).
- The Rendition module provides means for representing desired transformations on the events both on their dimensions and their content properties.

For the purpose of our comparison with HyTime HyQ query language [1], we consider here the main components of the Location address module.

Addressing on an FCS: An axis in HyTime is viewed as a sequence of units called *quanta*. An event on an axis can range over one or more quanta. A *range*, can be defined by a marker pair for a single axis, or a marker pair list for an FCS of more than one axis. As an example consider a single axis of 10 quanta, a dimension specification (dimspec) using the marker pair "1 2", would identify the first two quanta in the range. On an FCS with a pair of axes, an event ranging over the first two quanta of one axis and the first four quanta of the second axis would be represented with a list of marker pairs as follows : "1 2 1 4".

Addressing in a tree: When identifying an SGML element as a node in a tree, there is a need for a means of representing the *tree*. This can be done in four different ways. By using the :

- tree location address (`treeloc`) : a tree is considered as a list of levels, each level being a list of *nodes*. For instance, the node which is the first child of the second child of the sixth child of the root will be represented as "1 6 2 1" (e.g., the title of the first subsection of the second section of *my_article*, see the tree in Figure 4).
- list location address (`listloc`) : a tree is viewed as a list of nodes (e.g., an FCS with a single axis) visited by a depth first left to right traversal.



LEVELS \ PATHS	1	2	3	4	5	6	7	8	9	10
1	my_article	my_article	my_article	my_article	my_article	my_article	my_article	my_article	my_article	my_article
2	title	author	author	abstract	section	section	section	section	section	section
3					title	paragraph	figure	title	subsection	subsection
4									title	paragraph

Figure 4: HyTime paths in the specific logical structure of an article

- path location address (`pathloc`) : a tree is represented as a matrix (e.g., an FCS with two axes) where the rows distinguish the different levels of the tree and the columns contain all the paths from the root to the leaves. For instance, Figure 4 illustrates the matrix of *paths* in the specific logical structure of an article defined with the DTD of Figure 1. Thus, the dimension specification "9 1 1 -1" will be used to locate the title of the first subsection of the second section of *my_article* (in the extended O_2SQL this has been represented as $my\_article.sections[1].subsectns[0].title$).
- relative location address (`relloc`) : a tree is a set of nodes with genealogical relationships to each other. Given a node, another node is located relative to it by specifying its relationship (e.g., child) and a dimension specification that identifies it (e.g., "2 1" for the second child).

Finally, another addressing mechanism in the Location address module is Property location addressing (`proploc`). This module addresses the value of an SGML attribute or some other property (e.g., Generic Identifier) of an element by a qualified property name.

The HyQ Language

In this subsection we consider query facilities of HyQ [1] to compare them to the extended O_2SQL query language presented in Section 4. Due to space limitations, we do not present HyQ in detail (see [20]). HyQ is a query language designed to directly reflect HyTime constructs and addressing methods, namely HyTime Location address module facilities. We are only interested here in HyTime addressing facilities concerning SGML elements within a tree structure. Further encapsulation of SGML elements in events is beyond the scope of this paper. For comparing the extended O_2SQL with HyQ, we use the examples of queries presented in Section 4.

The semantics of HyQ is defined in terms of node lists on which a query operates. The domain of a query is defined by specifying the root node of the tree containing the source nodes. The query returns the list of nodes satisfying the related assertions. HyQ itself is a lisp-like language where queries are nested HyQ functions. From a structural point of view HyQ, compared to the extended O_2SQL, presents the following disadvantages :

a) It is a procedural query language (vs. declarative O_2SQL). As a result, the formulation of queries in HyQ is not user friendly.

b) It is based on an untyped data model (vs. O_2SQL with strong typing). We believe that typing is essential in particular for query optimization.

c) It allows elements to be retrieved from an SGML database. Nevertheless, complex data values cannot be constructed from the selected elements, as the result of a query (e.g., Query Q1 cannot be expressed in HyQ).

d) It uses paths to locate SGML elements in the document logical structure. However, conditions on paths mixed with conditions on the related elements cannot be expressed (e.g., Query Q1O cannot be expressed in HyQ).

The other queries presented in Section 4 can be expressed, but as mentioned in (a) their formulation is more complex. In the remainder of this subsection, we present the formulation in HyQ of some of these queries, to illustrate the difficulties of using this language. Consider Query Q2 :

```
Q2: <HyQ qdomain = my_article>
        select(DOMTREE
               EQ(proploc(CAND GI) "title"))
    </HyQ>
```

The body of a query is defined as the content of a special HyTime element type, called *HyQ*. The attribute *qdomain* identifies that the domain of the query is the root of a document, called *my_article*. The function `select()` takes a node list and an assertion and applies the assertion to each member of the node list in turn. This function corresponds to the basic O$_2$SQL filter `select-from-where`. The keyword DOMTREE (resp. DOMROOT) declares that the function `select()` is applied to all the nodes (resp. the root) of the tree representing *my_article*. The keyword CAND is used within the assertion to refer to the current candidate node. In Q2, the assertion is asking if, for a candidate node, the value of the property GI (Generic Identifier) is equal (`EQ()` function) to the string "title". The function `proploc()` is used to retrieve the related property. Note that *title* here is a data value whereas it is a data type for the extended O$_2$SQL, used during query evaluation to restrict search space (disadvantage (b)).

In order to express common queries that can be re-used HyQ allows the definition of query functions. These HyQ functions correspond to standard O$_2$SQL named queries [27] not presented in this paper. For instance, the following query function :

```
<HyQ fn=hasGI>
    select (%1 EQ(proploc(CAND GI) %2))
</HyQ>
```

returns the elements in a domain with a specified value for the attribute GI (e.g., a tag name). The attribute *fn* defines the name of the query (*hasGI*). The "%1" and "%2" are variable parameters for the domain and the specific generic identifier. A HyQ query function can be used in the sequel within another query using the function `useQ()` as follows :

```
Q2: <HyQ> useQ(hasGI DOMTREE "title") </HyQ>
```

Thus, `useQ()` is used to name the query function (*hasGI*) and pass the values of its arguments (DOMTREE and "title"). Elements of a document can be retrieved not only by their properties (e.g., SGML attributes) but also by their positions in the document specific structure (see Figure 4). Indeed, the function `pathloc()` is used to locate one or more paths down the tree representing a document in the domain of the query. For example, Query Q3 can be expressed as follows :

```
Q3: <HyQ qdomain=my_article>
        pathloc(DOMTREE (1 -1 2 2))
    </HyQ>
```

In Q3, `pathloc()` with the dimension specification "1 -1 2 2", first locates all the paths starting from the root of *my_article* (DOMTREE) and then locates the corresponding children and grandchildren (see Figure 4). In order to access tree nodes based on their genealogical relationships (e.g., child, ancestor, elder and younger sibling, etc.) to other nodes in the tree, HyQ provides a function `relloc()`. For instance, consider Query Q5:

```
Q5: <HyQ qdomain=my_article>
      useQ(hasGI
        relloc(useQ(hasGI DOMTREE "paragr")
               DOMROOT ESIB -1 1)
        figure")
    </HyQ>
```

In Q5, `relloc()` locates the last elder sibling (ESIB " -1 1") of the paragraphs selected by the inner *hasGI* query. Finally, the outmost *hasGI* query is applied in order to select the elder siblings which are figures. For a more intricate query let's now consider Query Q6:

```
Q6: <HyQ qdomain=my_article>
      listloc(useQ(hasGI
                relloc(listloc(useQ(hasGI
                                 DOMTREE "section")
                                 1 1)
                       DOMROOT DES (1 -1 2 -1))
                paragr")
             -1 1)
    </HyQ>
```

In Q6, the HyQ function `listloc()` is used to locate the first section ("1 1") from the list of sections selected by the inner query *hasGI*. Then, `relloc()` locates the descendants (DES "1 -1 2 -1") of the first section (e.g., all nodes in the paths from the source element to the bottom of the tree). Finally, `listloc()` is used once more in order to locate the last paragraph ("-1 1") from the list of paragraphs (descendants of the first section) selected by the outer query *hasGI*.

The following queries illustrate how cross references defined with the SGML addressing mechanism (ID/IDREF), can be queried using HyQ. The function `oo()` returns the elements (or entities) referenced in the body of the function, by their unique identifiers. For instance, consider Query Q7, where Articles is now a forest of trees representing documents of type article :

```
Q7: <HyQ qdomain=Articles>
      assign(paragr_ref
          proploc(select(useQ(hasGI
                           pathloc(useQ(hasGI
                                          DOMTREE
                                          "section")
                                     1 -1 2 1)
                           "paragr")
                      match(CAND "complex object")
          ATTVAL[reflabel])))
      select(nlref(paragr_ref)
          EQ(proploc(oo(CAND) GI) "figure"))
    </HyQ>
```

In Q7, pathloc() returns the children ("1 -1 2 1") of the sections selected by the inner query *hasGI*. Then, the query *hasGI* is used once more to select the children which are paragraphs. The function select() is used to return the paragraphs containing the string "complex object" (match() function). Afterwards, for the selected paragraphs, we assign (assign() function) a local name (*paragr_ref*) to the node list with the values of the corresponding attribute reflabel (proploc()). This node list can be referenced in the sequel by its name (*paragr_ref*) using the function nlref(). Finally, select() with domain the node list *paragr_ref*, returns the references of paragraphs that point (oo(CAND)) to a figure (proploc()). In the same spirit Query Q8 can be formulated as follows :

```
Q8:  <HyQ qdomain=my_article>
       assign(second_subsectn
              listloc(useQ(hasGI
                      pathloc(listloc(
                                useQ(hasGI
                                     DOMTREE
                                     "section")
                                     1 1)
                              (1 -1 2 1))
                      "subsectn")
              2 1))
     select(DOMTREE
              EQ(proploc(CAND ATTVAL[reflabel])
                 proploc(nlref(second_subsectn)
                                     ATTVAL[label])))
     </HyQ>
```

In Q8, we first assign a local name (*second_subsectn*) to the second subsection of the first section. Then we select all elements of *my_article* that have a reference (reflabel) to this subsection (EQ()) identified by its unique ID (label). In order to express the query Q9 we must use a nested select() function as follows :

```
Q9:  <HyQ qdomain=my_article>
       select(DOMTREE
              and(assign(cur_element CAND)
              select(pathloc(
                        listloc(useQ(hasGI
                                DOMTREE "section"))
                                1 1)
                     (1 -1 2 -1))
              and(EQ(proploc(CAND ATTVAL[label])
                     proploc(nlref(cur_element)
                                ATTVAL[reflabel])))))
     </HyQ>
```

In Q9, assign() is used to capture the value of the outmost select() candidate nodes (*cur_element*), in the domain of the query (*my_article*). Then, for each element contained in the first section of *my_article*, the inner select() compares (EQ()) its unique identifier (label) with the value of the attribute reflabel of the outmost select() candidate node (nlref(*cur_element*). Finally, the query returns the *cur_elements* where the assertion is true.

6 CONCLUSIONS

In this paper we use the model and the SQL-like query language of [10] in order to manage structured documents with hypertext links. The model represents a structured document with typed links as a complex object, and uses paths through the document structure, as first class citizens in formulating queries. Different data notations are manipulated with system or user defined methods for display, retrieval etc. Several examples of queries illustrate, from a practical point of view, the expressive power of the language to retrieve documents, even without an exact knowledge of their structure (e.g. tree, graph), in a simple and homogeneous fashion. In this way, the proposed language smoothly combines the features of both IRS and DBMS access languages with navigation facilities. In addition, it could serve as a back-end in order to build appropriate user interfaces for visual queries in the style of [5].

It must be stressed that the proposed model and language implement the equivalent HyTime Location Address Module. In fact, the extended O_2SQL is more powerful than the corresponding HyQ query language. We have already mentioned in the previous section advantages of the proposed SQL-like query language. Since the extended O_2SQL is a general purpose query language, is useful for a variety of OODBMS applications, whereas HyQ is application specific. Note also that the extended O_2SQL has formal foundations [10] that would be much more difficult to obtain with HyQ. Since HyQ is rather ill-defined, it must be subject to conflicting interpretations in different implementations.

The current implementation is based on SGML and the O_2 OODBMS, but the model and the query language are applicable for other OODBMS and standards. An extension of the Euroclid SGML parser [14] has been developed to translate SGML documents into O_2 schemas and instances. This extension requires the annotation of the BNF grammar generated by the parser, with semantic actions. It should be noted that the representation of SGML documents in an OODBMS such as O_2 comes with some extra cost in storage. This is the price to be paid for storing SGML fragments as individual objects. The extension of O_2SQL is also being implemented. Finally, the optimization of queries using path expressions with variables and the integration of appropriate pattern matching algorithms with full-text indexing mechanisms, are also currently being studied.

ACKNOWLEDGEMENTS

We are grateful to O_2Technology and Euroclid for their technical support during this project. We also thank M. Scholl, S. Abiteboul, S. Cluet, B. Amann and A. M. Vercoustre for helpful comments.

References

[1] ISO/IEC 10744. Information Technology- Hypermedia/Time-based Structuring Language (HyTime), 1992.

[2] ISO 8879. Information Processing- Text and Office Systems- Standard Generalized Markup Language (SGML), 1986.

[3] R. Akscyn, D. McCracken, and E. Yoder. KMS: A Distributed Hypermedia System for Managing Knowledge in Organizations. *Comm. of the ACM*, 31(7):820–835, July 1988.

[4] B. Amann and M. Scholl. GRAM: A Graph Model and Query Language. In *ECHT'92*, pages 201–211. ACM, December 1992.

[5] B. Amann, M. Scholl, and A. Rizk. Querying Typed Hypertexts in Multicard/O_2. In *ECHT'94*, Edinburgh, England, September 1994. ACM.

[6] F. Bancilhon, S. Cluet, and C. Delobel. A Query Language for the O_2 Object-Oriented Database System. In *DBPL'89*, pages 122–138, Salishan Lodge, Oregon, June 1989. Morgan Kaufmann.

[7] C. Beeri and Y. Kornatzky. A Logical Query Language for Hypertext Systems. In *ECHT'90*, pages 67–80, Versailles France, November 1990. INRIA.

[8] AIS Berger-Levrault. SGML/Search, Description du Langage. Internal Document, 34 Avenue du Roule 92200 Neuilly sur Seine, 1993.

[9] F. J. Burkowski. An Algebra for Hierarchically Organized Text-Dominated Databases. *Information Processing & Management*, 28(3):333–348, 1992.

[10] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From Structured Documents to Novel Query Facilities. In *SIGMOD'94*, pages 313–324, Minneapolis, Minnesota, May 1994. ACM.

[11] M. P. Consens and A. O. Mendelzon. Expressing Structural Hypertext Queries in GRAPHLOG. In *Hypertext'89*, pages 269–292, November 1989.

[12] O. Deux. The Story of O_2. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):91–108, Mars 1989.

[13] EBT. DynaText. Product Reference Manuel, 1994.

[14] Euroclid. Le Parseur SGML d'Euroclid. Internal Document, 12, Avenue des Prés 78180 Montigny le Bretonneux, 1991.

[15] F. Garzotto, P. Paolini, and D. Schwabe. HDM: A Model for the Design of Hypertext Applications. In *Hypertext'91*, pages 313–328, San Antonio, Texas, December 1991. ACM.

[16] C. F. Goldfarb. *The SGML Handbook*. Clarendon Press, Oxford, 1990.

[17] M. Gyssens, J.Paredaens, and D. V. Gucht. A Graph-Oriented Object Database Model. In *PODS'90*, pages 417–424, Nashville Tennesee, April 1990. ACM.

[18] F. Halasz, T. Noran, and R. Trigg. NoteCards as a Nutshell. In *Conf. of Humain Factors in Computing Systems*, pages 45–52, Toronto, Canada, April 1987. ACM SIGCHI + GI.

[19] F. G. Halasz. Reflections on NoteCards: Seven Issues for the Next Generation of Hypermedia Systems. *Comm. of the ACM*, 31(7), July 1988.

[20] W. Eliot Kimber. HyTime and SGML: Understanding the HyTime HyQ Query Language. Availiable via anonymous ftp at ftp.ifi.uio.no /pub/SGML/HyTime, August 1993.

[21] D. Lucarella, S. Parisotto, and A. Zanzi. MORE: Multimedia Object Retrieval Environment. In *Hypertext'93*, pages 39–50, Seattle, Washington USA, November 1993. ACM.

[22] I. A. Macleod. Storage and Retrieval of Structured Documents. *Information Processing & Management*, 26(2):197–208, 1990.

[23] J. Nanard and M. Nanard. Using Structured Types to Incorporate Knowledge in Hypertext. In *Hypertext'91*, pages 329–343, San Antonio, Texas, December 1991. ACM.

[24] V. Quint and I. Vantton. Combining Hypertext and Structured Documents in Grif. In *ECHT'92*, pages 23–32. ACM, December 1992.

[25] P. D. Stotts and R. Furuta. Hypertext 2000: Databases or Documents? *Electronic Publishing*, 4(2):119–121, June 1991.

[26] P. D. Stotts, R. Furuta, and J. C. Ruiz. Hyperdocuments as Automata: Trace-based Browsing Property Verification. In *ECHT'92*, pages 272–281. ACM, December 1992.

[27] O_2 Technology. *The O_2 User's Manual, version 4.4*, December 1993.

[28] A. M. Vercoustre. Structured Editing - Hypertext Approach: Cooperation and Complementarity. In *EP'90*, pages 65–78. Cambridge University Press, September 1990.

[29] P. T. Zellweger. Scripted Documents: A Hypermedia Path Mechanism. In *Hypertext'89*, pages 1–14, ACM, November 1989.