From Structured Documents to Novel Query Facilities*

V. Christophides, S. Abiteboul, S. Cluet and M. Scholl†

I.N.R.I.A.
78153 Le Chesnay, Cedex
France
(† Cedric/CNAM, 292 rue St Martin 75141, Paris Cedex 03, France)
{Vassilis.Christophides, Serge.Abiteboul, Sophie.Cluet, Michel.Scholl}@inria.fr

February 29, 1996

Abstract

Structured documents (e.g., SGML) can benefit a lot from database support and more specifically from object-oriented database (OODB) management systems. This paper describes a natural mapping from SGML documents into OODB's and a formal extension of two OODB query languages (one SQL-like and the other calculus) in order to deal with SGML document retrieval.

Although motivated by structured documents, the extensions of query languages that we present are general and useful for a variety of other OODB applications. A key element is the introduction of paths as first class citizens. The new features allow to query data (and to some extent schema) without exact knowledge of the schema in a simple and homogeneous fashion.

1 Introduction

Structured documents are central to a wide class of applications such as software engineering, libraries, technical documentation, etc. They are often stored in file systems and document access tools are somewhat limited. We believe that (object-oriented) database technology (OODB) can bring a lot of benefit to documents management, e.g., recovery, concurrency control and high level query capabilities. In particular, whereas existing tools for accessing documents are sophisticated with respect to pattern matching facilities, they do not support queries on non textual data, often totally ignore document structure, and are extremely primitive from a *logical viewpoint*. For these reasons, we introduce extensions of an OODB data model and of two query languages (one SQL-like and the other calculus) to fit the needs of structured document storage and retrieval. The main contribution of the paper resides in novel language features that are important beyond the scope of structured documents.

The Standard General Mark-up Language (SGML) [2] is becoming *de facto* the standard for structured document creation and exchange. An SGML document has a hierarchical structure satisfying a document type definition (DTD). In the spirit of [4], we design a mapping from

*Work partially supported by Esprit BRA *FIDE2*.

DTD's to OODB schemas[1]. It will be shown that, although straightforward, this mapping requires structuring primitives such as ordered tuples, lists, and union types. To the best of our knowledge none of the existing OODB management systems (OODBMS) provide all the needed features. We choose as a target system the O_2 OODBMS [15] for its type system and moreover for its elegant query language O_2SQL [8]. The first part of our work consists of an appropriate extension of the O_2 data model in order to facilitate the mapping from SGML documents to O_2 instances.

Once in the database, documents can be queried like other data. However, standard query languages lack features that are essential for documents retrieval such as pattern matching facilities and querying data without exact knowledge of its precise structure. We introduce appropriate extensions of OODB query languages to answer these needs as well as those induced by the new structuring primitives. The new features lead into a number of subtle issues, in particular, with respect to typing. We demonstrate how these can be resolved by formally presenting an extension of the OODB calculus underlying IQL [6].

The most interesting novelty (from a technical viewpoint) comes from the use of *paths* (to navigate through the database objects/values). In the language, paths are first class citizens and in particular, path variables may be used in queries. We will see how these new features can be incorporated in O_2SQL allowing to query documents (even without precise knowledge of their structure) in a simple and homogeneous fashion. For instance, we will show how to obtain the "difference" between the structures of two documents with a short and very intuitive query.

Recently, many researchers have studied the modeling of document databases [14, 19, 25, 27]. These models provide only some of the features that, as will be seen in the sequel, are critical for the representation of SGML documents. Additionally, several query languages for structured documents can be found in the literature [18, 20, 26, 11, 9]. As opposed to these approaches, the language we propose (i) can be applied in the general framework of OODB applications and (ii) allow queries that address both the content and the structure of the documents. The OODB languages presented in [10, 24] like our language allow to query data with incomplete knowledge of its structure. Compared to these, our language has simpler formal foundations and does not consider paths as strings but as first-class citizens that can be queried. Finally, the use of paths relates our language to query languages proposed for graph databases or hypertext systems, e.g., [13, 7]. Indeed, we believe that our language is particularly suited for current extensions of SGML to multi and hypermedia documents such as HyTime [1].

This paper is organized as follows. Section 2 introduces the SGML standard. The mapping from SGML to the O_2 DBMS is defined in Section 3. Section 4 presents the extension of the O_2SQL language and Section 5 the formal bases for this extension. Section 6 concludes the paper and briefly presents the status of the implementation.

2 SGML preliminaries

In this section, we present the main features of SGML [2, 17]. (A general presentation is clearly beyond the scope of this paper.)

In order to define a document's logical structure, SGML adds descriptive mark-up (tags) in document instances. Each SGML document has: (i) A prologue including a Document Type Definition (DTD), i.e., a set of grammar rules specifying the document *generic logical structure* (see Figure 1); and (ii) a document instance containing the information content as well as the tags e.g., the *specific logical structure* of the document (see Figure 2).

[1]The inverse mapping from database schema/instances to SGML DTD/documents also opens interesting perspectives for exchanging information between heterogeneous databases, writing reports, etc. This is not considered

```
 1. <!DOCTYPE    article [
 2. <!ELEMENT   article      - -      (title, author+, affil, abstract, section+, acknowl)>
 3. <!ATTLIST    article              status   (final | draft)   draft>
 4. <!ELEMENT   title        - -      (#PCDATA)>
 5. <!ELEMENT   author       - O      (#PCDATA)>
 6. <!ELEMENT   author       - O      (#PCDATA)>
 7. <!ELEMENT   abstract     - O      (#PCDATA)>
 8. <!ELEMENT   section      - O      ((title, body+) |  (title, body*, subsectn+))>
 9. <!ELEMENT   subsectn     - O      (title, body+)>
10. <!ELEMENT   body         - O      (figure | paragr)>
11. <!ELEMENT   figure       - O      (picture, caption?)>
12. <!ATTLIST    figure               label   ID   #IMPLIED>
13. <! ELEMENT   picture     - O      EMPTY>
14. <!ATTLIST    picture              sizex    NMTOKEN  "16cm"
                                      sizey    NMTOKEN  #IMPLIED
                                      file    ENTITY #IMPLIED>
15. <!ELEMENT   caption     O O       (#PCDATA)>
16. <!ENTITY     fig1     SYSTEM  "/u/christop/SGML/image1"   NDATA >
17. <!ELEMENT   paragr      - O       (#PCDATA)>
18. <!ATTLIST    paragr               reflabel   IDREF   #REQUIRED>
19. <!ELEMENT   acknowl  - O          (#PCDATA)>           ]>
```

Figure 1: A DTD for a document of type article

In SGML jargon, the logical components of a document are called *elements*. For example line 2 of Figure 1 defines the structure of the element with name article. Element names are used as tags in the document. The specification of an element in the DTD gives its name, its structure and some indications (e.g., "-O" indicates that the tag can be omitted if there is no ambiguity). The element structure is built using other elements or basic types such as #PCDATA, EMPTY, etc. and connectors that 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 figure is composed of a picture followed by a caption (line 11). There is also an alternative aggregation connector ("&") that does not imply an order.
- The choice connector ("|") provides an alternative in the type definition. For instance, element body is either a figure or a paragraph (line 10).
- The optional indicator ("?") indicates zero or one occurrence of an element (e.g., captions in figures (line 11)); the plus sign ("+") indicates one or more occurrences of an element (e.g., sections in articles, line 2); and the asterisk ("*") zero or more occurrences.

There are a number of other features in SGML that we do not present here because of space limitations. We conclude by mentioning one important modeling feature. Cross references between elements are specified using keywords ID (for the element that will be referenced) and

in the present paper.

```
<article status="final">
<title> From From Structured Documents to Novel Query Facilities </title>
<author> V. Christophides
<author> S. Abiteboul
<author> S. Cluet
<author> M. Scholl
...
<abstract> Structured documents (e.g., SGML) can benefit a lot from
database support and more specifically from object-oriented database
(OODB) management systems...
<section>
<title> Introduction </title>
...
<body><paragr> This paper is organized as follows. Section 2
introduces the SGML standard. The mapping from SGML to the O2
DBMS is defined in Section 3. Section 4 presents the extension ...
</body></section>
<section>
<title> SGML preliminaries </title>
<body><paragr> In this section, we present the main features of
SGML. (A general presentation is clearly beyond the scope of this paper.)
</body></section>
...
</article>
```

Figure 2: An SGML document of type article

IDREF (for the element referencing it). For instance, figures may be referenced in paragraphs (Figure 1 lines 12 and 18).

3 From SGML to O_2

In this section, we consider the representation of SGML documents in an OODB, namely O_2. We show briefly what portions of the document model are easily handled by the database model and the extensions that are needed to support the rest. We consider the problem of translation from documents to database instances.

None of the existing OODBMS's provides all the features we need. We chose O_2 [15] because of (i) its sophisticated type system and (ii) its query language that can easily be extended. The SGML hierarchical structure can be naturally represented using the complex values in O_2. Indeed, O_2 has an hybrid data model including objects and constructed values. The modeling would be more complicated in a pure object model. The aggregation and list constructors of O_2 are very useful to describe the structure of elements.

Documents possibly require multimedia types (e.g., images) that can easily be incorporated into any OODB and cross references between logical components that are easily modeled using object identity. Furthermore, object sharing provided by OODB's is useful for the manipulation of documents, notably in the support of their evolution (e.g., versions).

On the other hand SGML document modeling leads to other requirements mainly related to the use of connectors and occurrence indicators mentioned in Section 2 that are missing in O_2 and

```
class Article public type tuple (title: Title, authors: list (Author), affil: Affil,
                                 abstract: Abstract, sections: list (Section),
                                 acknowl: Acknowl, 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, bodies: list (Body)),
                                 a2: tuple (title: Title, bodies: list (Body),
                                        subsectns: list (Subsectn)))
          constraint:           (a1.title != nil, a1.bodies != list())
                                 | (a2.title !=  nil, a2.subsectns != list())
class Subsectn public type tuple (title: Title, bodies: list (Body))
          constraint: title != nil, bodies != list()
class Body public type union (figure: Figure, paragr: Paragr)
          constraint: figure != nil | paragr != nil
class Figure public type tuple (picture: Picture, caption: Caption,
                                private label: list (Object))
          constraint: picture != nil
class Picture inherit Bitmap
class Caption inherit Text
class Paragr inherit Text
          public type tuple (private reflabel: Object)
          constraint:  reflabel != nil
class Acknowl inherit Text
name Articles: list (Article)
```

Figure 3: O_2 Classes for Documents of Type Article

in other OODB's as well. In particular, two main modeling features lead to extensions of the data model:

Ordered tuples. The ordering of tuple components is meaningful in SGML. Sometimes, the ordering is imposed; sometimes some flexibility is left. We propose an original solution based on polymorphism that blurs the distinction between tuples and lists. A more ad-hoc solution based on adding to the tuple an attribute recording the ordering between the fields would lead to inelegance in the query language.

Union of types. In SGML, alternative structures may be provided for the same element type. Union of types is the appropriate solution for this. We felt that type union is a useful modeling feature that was missing in O_2 and added it to our model. We use *marked* union, i.e., a tag specifies the alternative type that is chosen. Union (generalization) could also be simulated using multiple inheritance but we did not adopt this solution that misuses the original inheritance semantics (specialization). In addition, this latter solution would needlessly increase the size of the class hierarchy.

We will see in Section 5 how these features can formally be added to the model.

The problem of extracting data from sequential structured files has been quite popular lately (see for example [28, 4, 21]). More specifically, our problem consists in getting an O_2 database representation from an SGML document. In other words, we need to map a DTD into an O_2 schema and a document instance into corresponding objects and values. In the spirit of [4], we propose to do this by annotating SGML DTD's (or some intermediate BNF grammar) with semantic actions. This is quite standard in the compilation field, see for example [22]. Each SGML element definition in the DTD is 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 3 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 content type (e.g., `Text`, `Bitmap`). The O_2 tuples should be viewed as ordered. The choice connector ("|") is modeled by a union type (e.g., class Body). For unnamed SGML elements defined through nested parentheses (e.g., (title, body+) line 7, Figure 1), system supplied names are provided (e.g., a_1 in class Section, Figure 3). Observe the use of inheritance (in class Picture) and that of private attributes (e.g., status in class Article). The element components marked by a "+" or "*" occurrence indicator are represented by lists (e.g., attribute authors in class Article). Finally, note that constraints had to be introduced to capture certain aspects of occurrence indicators, the fact that some attributes are required and also the range restrictions. Constraints will not be considered in this paper.

To conclude this section, it should be noted that the representation of SGML documents in an OODB such as O_2 comes with some extra cost in storage. This is typically the price paid to improve access flexibility and performance.

4 The Query Language

In this section, we present an extension of the O_2SQL language. The presentation relies on examples. The running example uses the O_2 schema of the previous section. The formal foundations are considered in Section 5.

In the SGML world, documents are usually queried by means of Information Retrieval Systems (IRS). These systems provide two main facilities lacking in OODB query languages: (i) IRS's provide sophisticated pattern matching facilities for selecting documents according to their content relying on full text indexing; and (ii) IRS's do not require users to know the exact structure of documents. Obviously, if we intend to query documents from a database, we must provide such facilities in the richer model of the previous section. To answer (i), we show how sophisticated string predicates are introduced in the language. Then, to answer (ii), we introduce two new sorts (PATH and ATT) and their basic operations. Finally, we show how the extensions of the model impact the query language. In particular, we show how union type affects the O_2SQL language. This leads to a language that combines (we believe nicely) the features of both IRS and database access languages.

4.1 Querying Strings

The semantics of O_2SQL relies on a functional approach. O_2SQL is defined by a set of basic queries and a way of building new queries through composition and iterators. Thus, to add a new functionality to the language one often has only to add new basic queries. This is what we do here.

The next query introduces the predicate `contains` which offers pattern matching facilities.

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

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

The `contains` predicate allows to match a string with a pattern or a boolean combination of patterns. (Patterns are constructed using concatenation, disjunction, Kleene closure, etc.) Among other textual predicates, we only cite `near` that allows to check whether two words are separated by, at most, a given number of characters (or words) in a sentence.

From a linguistic viewpoint, this raises no issue. On the other hand, the integration of appropriate pattern matching algorithms and full text indexing mechanisms are interesting technical issues that we are currently studying. This is clearly beyond the scope of the present paper.

4.2 Managing Union Types

The introduction of union types involves some serious modification of the type checking of O_2SQL queries as well as of the evaluation of O_2SQL iterators (`select-from-where`, `exists`, etc). We first explain the typing mechanism.

O_2SQL imposes typing restrictions. For instance, when constructing a collection, we check that its elements have a common supertype (e.g., sets containing integers and characters are forbidden). Since we introduce union types in the model, we have to specify their subtyping rules. We briefly present the two main rules here.

1. There is no common supertype between a union type and a non union type. Note that this forbids, for instance, to perform an intersection between a set of integers and a set of (a:integer + b:char)'s.
2. Two union types have a common supertype if they do not have an attribute (marker) conflict (i.e., types with the same attribute a whose domains do not have a common supertype). When it exists, the (least) common supertype of two union types is the union of the two types. For instance, (a:integer + b:char + c:string) is the least common supertype of (a:integer + b:char) and (b:char + c:string). Note that this second typing rule may result into a combinatorial explosion of types. However, (i) this should rarely happen and (ii) some semantic rules can be added to the O_2SQL typing mechanism in order to control this inflation.

The following example shows how the O_2SQL evaluation of iterators is modified to take into account union types.

Q2: *Find the subsections of articles containing the sentence "complex object".*

```
select  ss
from    a in Articles, s in a.sections,
        ss in s.subsectns
where   text(ss) contains ("complex object")
```

Compared to Q1, the `contains` operation in query Q2 is evaluated not over individual data objects but over complex logical objects (e.g., subsectns). For that we use a system supplied operator `text` performing an inverse mapping from a logical object (e.g., a subsectn) to the corresponding portion of text [5].

Recall now that, in the schema of Figure 3, sections have a union type: a section marked with a_1 corresponds to a tuple with attributes title and bodies and a section marked with a_2, to a tuple with attributes title, bodies and subsectns. In query Q2, the variable ss ranges over subsectns of sections marked with a_2. Two remarks are noteworthy. First, in the definition of variable ss in the `from` clause, the a_2 marker is omitted. This syntactic sugaring is required because users are not always aware of markers in union types. Second, we do not want the evaluation to fail on sections whose instance type does not correspond to the a_2 marker (e.g., sections that do not have subsectns).

To deal with this problem, we introduce the notion of *implicit selectors.* Any operation on a variable ranging over a domain with union type implies an implicit selection. In the above query, the implicit selection is that s.a_2 should be defined. It must be noted that this mechanism stands only for variables and not for named instances. For example, suppose the existence of the name *my_section* which has been instantiated with a section marked with a_1. In this case, the query *my_section.subsectns* will return a type error detected at execution time.

Let us now come back to the syntactic sugaring with a more complex example. Suppose that, instead of the subsections, we are now interested by the bodies of articles. The `from` clause of the query becomes:

`from` a `in` Articles, s `in` a.sections, b `in` s.bodies.

Variable b will range over the union of $s.a_1.bodies$ and $s.a_2.bodies$ and two implicit selectors will be used to avoid failure at evaluation. In this example, both attributes *bodies* have the same type. However, this cannot be guaranteed in all cases. When this is not the case, a marked union type is generated by the system.

4.3 Querying Paths

In order to query data without exact knowledge of its structure we introduce two new sorts: PATH and ATT. A value of the former denotes a *path* through complex objects/values (crossing objects, tuples, unions, lists, etc.) and a value of the latter represents an attribute name (of a tuple or a marked union). For instance, *.sections[0].subsectns[0]* is a path selecting the attribute sections of type list, then the first section, the attribute subsectns of this section and finally the first subsection. *Paths can be queried like standard data.* The following queries illustrate the use of path querying.

Like all types manipulated in O_2SQL, PATH and ATT come with their variables and basic queries. To distinguish variables according to their sort (standard data, PATH or ATT), PATH variables are prefixed by "$PATH\_$" and ATT variables by "$ATT\_$". In the next query, *my_article* is a root of persistence representing an article. Consider the query:

Q3: *Find all titles in my_article.*

```
select  t
from    my_article PATH_p.title(t)
```

The result is a set of titles reached by following various paths in the article structure (general title, title of each section, subsection, etc.).

In query Q3, the subquery *my_article PATH_p.title(t)* illustrates a new basic query. It is a path expression with variables (here $PATH\_p$ and t), whose semantics is different from that traditionally used in path expressions without variables. 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 the path values 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 path *PATH_p*. Several points need further comments.

1. We may allow the syntactical sugared form
   `from` *my_article .. title(t)*
   if we are not interested in the actual values of path variables.

2. Note that the presence of path variables will often imply that the corresponding data variable is of a union type. Indeed, following different paths, one should expect reaching different types. This is particularly true if we query data from different sources, e.g., various authors may structure sections differently.

3. Path variables may be used outside a `from` clause without being defined in such a clause. For instance, the expression
   my_article PATH_p.title
   is a query that returns the set of paths to a title field.

4. 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 path of value *.sections[0].subsectns[0]* we can compute the length of P: `length`$(P) = 4$ and project P on its first two elements: $P[0:1]$ = .sections[0].

5. When path variables are used in a path expression, there is always the possibility of cycles (in the schema and in the data). Our interpretation avoids cycles as will be shown in the next section.

To see another example of the use of paths for querying data with incomplete knowledge of their structure, let us assume the existence of a name *my_old_article* representing an old version of *my_article*. Consider the query:

Q4: *Find the structural differences between two versions of my_article.*

my_article PATH_p − *my_old_article PATH_p*

The left (resp. right) argument of the difference operation returns the set of paths starting from *my_article* (resp. *my_old_article*). Thus, the difference operation will return the paths that are in the new version of *my_article* and not in the old one. Supplementary conditions on data would allow the detection of possible updates or moves of individual textual objects within the document logical structure.

In a last example, we also use attribute variables:

Q5: *Find the attributes defined in my_article whose value contains the string "final".*

```
select name(ATT_a)
from   my_article PATH_p.ATT_a(val)
where  val contains ("final")
```

In this example, the data variable *val* ranges over data that can be accessed from *my_article* following a path denoted by *PATH_p* (which possibly is the empty path) and ending with the attribute denoted by the variable *ATT_a*. Accordingly, the initial type of *val* is the union of all the types found in the structure of *my_article*. The subquery *val* `contains` *("final")* uses the implicit selector *val.a* where a : *string* represents an attribute of type string in the corresponding union type. As a result O_2SQL restricts *val* to type *string*. The returned attributes are those whose value contains "final". We believe that this is an important feature of the extension of O_2SQL allowing the user to perform search operations like Unix `grep` inside an OODBMS. Finally, the function `name` returns a string, the name of an attribute.

4.4 Querying Attributes on their Position

In this last section, we consider the problem of ordered tuples. We have seen that the tuples used in the representation of documents are ordered. This implies that there is another way of viewing a tuple: as an heterogeneous list. For instance, we suppose that our database contains letters with a preamble including, the recipient (attribute *to*) and the sender (attribute *from*) addresses, in permutable order (i.e., introduced by the SGML "&" connector, see Section 2). Consider the query:

Q6: *Find the letters where the sender precedes the recipient in the preamble.*

```
select  letter
from    letter in Letters, letter.preamble[i].to,
        letter.preamble[j].from
where   i < j
```

In this query, we consider the tuple [to: string, from: string, ...] representing the preamble of a letter as a list of elements belonging to a union type [(to: string + from: string + ...)], each of the types being marked by the attributes name of the original tuple. Thus, the `from` clause of the query defines three data variables: *letter* that ranges over the letters, i and j that range respectively over the positions of markers *from* and *to* in the corresponding letters. The `where` clause is used to select the correct triples (letter,i,j) and the `select` clause returns the corresponding letters.

5 The Formal Model

In this section, we reexamine the issues found in previous sections by formalizing the data model.

5.1 Preliminaries

For the data structure, we use the formalism of IQL [6] and O_2 [23] with two notable distinctions: (i) marked union is introduced; and (ii) tuples are ordered. The new material is highlighted with boxes.

To start, we need a number of atomic types and their pairwise disjoint corresponding domains: **integer, string, boolean, float**. The set **dom** of atomic values is the union of these domains. We also need an infinite set **att** of attributes names $(a_1, a_2, \ldots)$; an infinite set **obj** of object identifiers also called oids $(o_1, o_2, \ldots)$; and a set **class** of class names $(c_1, c_2, \ldots)$. A special name *nil* represents the undefined value. Given a set O of oids, a *value* over O is defined by: (i) *nil*, and each element of **dom** or O, are values over O; (ii) if $v_1, \ldots, v_n$ are values, and $a_1, \ldots, a_n$ distinct attributes, the tuple $[a_1 : v_1, \ldots, a_n : v_n]$, the set $\{v_1, \ldots, v_n\}$ and the list $[v_1, \ldots, v_n]$ are values.

The set of all values over O is denoted **val**(O). An *object* is a pair (o, v) where o is an oid and v a value. Observe that since we consider ordered tuples, for each permutation $i_1, \ldots, i_n$ of $1, \ldots, n$, which is not the identity, $\boxed{[a_1 : v_1, \ldots, a_n : v_n] \neq [a_{i_1} : v_{i_1}, \ldots, a_{i_n} : v_{i_n}].}$

Typing is an essential component in our framework. Typing is defined with respect to a given set C of classes[2]. The *types* over C, denoted **types**(C), are defined as follows:

[2] We follow here the IQL or O_2 tradition. A class is a typing notion that should not be confused with the *class extension*, the set of objects of that class. So, the class hierarchy is a hierarchy of types that is not the class extension hierarchy.

1. **integer, string, boolean, float,** are types;

2. the class names in C are types; **any** (the top of the class hierarchy) is a type;

3. if τ is a type, $[\tau]$ and $\{\tau\}$ are types (resp. list and set types);

4. if $\tau_1, \ldots, \tau_n$ are types and $a_1, \ldots, a_n$ attribute names, $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ is a (tuple) type;

5. if $\tau_1, \ldots, \tau_n$ are types and $a_1, \ldots, a_n$ attribute names, $\boxed{(a_1 : \tau_1 + \ldots + a_n : \tau_n) \text{ is a (union) type.}}$

Note that we allow marked union a feature not found in the standard O_2 data model. A tuple of the form $[a_i : v]$ (for i in $[1..n]$) where v is of type τ_i is a value of the type $(a_1 : \tau_1 + \ldots + a_n : \tau_n)$. Another extension of the O_2 data model is that the ordering of tuple attributes is meaningful. This is to capture the use of such ordering in, for instance, SGML. It can be easily ignored for applications where this ordering makes no sense.

Inheritance is important in the OODB context (and perhaps less so in the SGML context). We describe it next insisting only on the non-standard portion.

A *class hierarchy* is a triple $(C, \sigma, \prec)$ where C is a finite set of class names, σ a mapping from C to **types**(C), and $\prec$ a partial order on C. The sub-typing relationship ($\leq$) is defined as in O_2 except that we add two rules. The first one deals with union:

$$[a_i : \tau_i] \leq (\ldots + a_i : \tau_i + \ldots).$$

Observe that as a consequence of the sub-typing rules, we now have that:

$$[a_1 : \tau_1, \ldots, a_n : \tau_n] \leq [a_i : \tau_i] \leq (a_1 : \tau_1 + \ldots + a_n : \tau_n).$$

Furthermore, we introduce a second new rule to view a tuple as a special case of heterogeneous list:

$$\boxed{[a_1 : \tau_1, \ldots, a_n : \tau_n] \leq [(a_1 : \tau_1 + \ldots + a_n : \tau_n)].}$$

This allows us to blur the distinction between a tuple (e.g., $[A : 5, B : 6]$) and the corresponding heterogeneous list (e.g., $[[A : 5], [B : 6]]$). We only consider *well-formed class hierarchies*, i.e., hierarchies $(C, \sigma, \prec)$ such that for each c, c', if $c \prec c'$, then $\sigma(c) \leq \sigma(c')$. We give now the semantics of classes and types.

Definition Let $(C, \sigma, \prec)$ be a class hierarchy. A *disjoint oid assignment* is a function π_d mapping each name in C to a disjoint finite set of oids. The *oid assignment* π (inherited from π_d) is given by: for each c, $\pi(c) = \cup\{\pi_d(c') \mid c' \prec c\}$. $\Box$

The syntax and semantics of types are now defined for a given class hierarchy and an oid assignment π. Given an oid assignment π, the *interpretation* of a type τ, denoted $dom(\tau)$, is given by:

- for each atomic type, take its corresponding domain;
- $dom(\textbf{any})$ is $\cup\{\pi(c) \mid c \in C\}$;
- for each $c \in C$, $dom(c) = \pi(c) \cup \{nil\}$.
- $dom(\{\tau\}) = \{\{v_1, \ldots, v_j\} \mid j \geq 0, \text{ and } v_i \in dom(\tau), i = 1, \ldots, j\}$,
- $dom([\tau]) =$
  $\{[v_1, \ldots, v_j] \mid j \geq 0, \text{ and } v_i \in dom(\tau), i = 1, \ldots, j\}$,

- $dom([a_1:\ \tau_1,\ldots,a_k:\ \tau_k]) =$
  $\{[a_1:\ v_1,\ldots,a_k:\ v_k,\ldots,a_{k+l}:\ v_{k+l}] \mid v_i \in dom(\tau_i), i=1,\ldots,k; l \geq 0\}$.
- $\boxed{dom(a_1:\ \tau_1 + \ldots + a_k:\ \tau_k) = \bigcup\{dom([a_i:\ v_i]) \mid 1 \leq i \leq k\}.}$

By abuse of notation, we denote by *dom* the mapping which associates to each type τ the set of ($\equiv$) equivalence classes of the elements in $dom(\tau)$. Then one can show that for each τ, τ' in **types**(C), if $\tau \leq \tau'$, $dom(t) \subseteq dom(t')$. For instance, to blur the distinction between a tuple and the corresponding heterogeneous list, consider the equivalence relation obtained with: $[a_1:\ v_i,\ldots,a_k:\ v_k] \equiv [[a_1:\ v_i],\ldots,[a_k:\ v_k]]$ for each tuple $[a_1:\ v_i,\ldots,a_k:\ v_k]$. It must be stressed that the typing mechanism of the extended O_2SQL (see Section 4.2) relies on the above subtyping rules.

To conclude these preliminaries, we present schemas and instances and comment on their definitions. Our schema does include methods in the style of O_2 but we do not discuss methods here and introduce them just for the sake of completeness.

Definition A *schema* S is a 5-tuple $(C, \sigma, \prec, M, G)$ where $(C, \sigma, \prec)$ is a well-formed class hierarchy; M is a set of method signatures; and G is a set of names (the roots of persistence) with a type $type(g)$ associated with each name g in G. $\Box$

Definition An *instance* I of schema $(C, \sigma, \prec, M, G)$ is a 4-tuple (π, ν, μ, γ), where (i) π is an oid assignment and $O = \cup\{\pi(c) \mid c \in C\}$; (ii) ν maps each object to a value in **val**(O) of correct type (i.e., for each c, and $o \in \pi(c)$, $\nu(o) \in dom(\sigma(c))$); (iii) μ assigns a semantics to method names in agreement with the types of methods given in M; and (iv) γ associates to each name in G of type τ, a value in $dom(\tau)$. $\Box$

5.2 The Calculus

We define a many sorted calculus in the spirit of [3]. The sort issue is quite intricate and we will come back to this in Subsection 5.3. For the time being, we only distinguish between three sorts: **val, att** and **path**. All variables have one of these sorts. The main additions are the introduction of attribute and path variables to provide more flexibility in querying data without precisely knowing its structure. Therefore most of our presentation is devoted to presenting and discussing path expressions.

As we have seen in Section 4 paths allow to navigate within database objects/values. Within a data value, if we are located at a tuple or a marked union, we may follow an attribute selector[3] ($\cdot a$ below); if we are at a list, we may choose an element of the list ($[i]$ below); if we are at a set, we may choose an element of the set ($\{v\}$ below). Finally, if we are at an object (and so reached the frontier of a value), we may use dereferencing ($\rightarrow$ below). Formally, a path variable will be interpreted using *concrete paths* which are sequences of:

1. "$\cdot a$" where a is an attribute name;
2. $[i]$ where i is an integer;
3. $\rightarrow$ (dereferencing);
4. $\{v\}$ where v is a value.

Several examples of concrete paths can be found in Subsection 4.3.

[3] Although not done here it is also possible to consider paths that goes though method calls.

The set of all concrete paths is denoted **path**. Observe that as defined here a path is allowed to cross the boundary of the value of an object; as a consequence, it may be the case that the number of concrete paths in a database is infinite because of the existence of cycles. Our semantics will enforce that only a finite number of concrete paths will be considered when navigating from a given value.

We denote variables with capital letters. We use three disjoint alphabets of variables: (i) data variables (X, Y, Z possibly with subscripts); path variables (P, Q, R possibly with subscripts); attribute variables (A, B, C possibly with subscripts). We next define attribute terms, path terms and data terms.

An *attribute term* is either an attribute name or an attribute variable. Now, *path terms* are given by:

1. each path variable is a path term;
2. ϵ (the empty string) and $\rightarrow$ are path terms;
3. if A is an attribute term, then $\cdot A$ is a path term;
4. if i is an integer term[4], then $[i]$ is a path term;
5. if P, Q are path terms and X is a data variable, then PQ, $P(X)$, $P\{X\}$ are path terms.

Finally, we define the *data terms* which are given by:

1. each name in G, each atomic value (nil, elements of **dom** or O), and each data variable are data terms;
2. if $t_1, \ldots, t_n$ are data terms and $A_1, \ldots, A_n$ attribute terms, then $[t_1, \ldots, t_n]$, $[A_1 : t_1, \ldots, A_n : t_n]$ and $\{t_1, \ldots, t_n\}$ are data terms;
3. if $t_1, \ldots, t_n$ are data terms and m a method in M, then $m(t_1, \ldots, t_n)$ is a data term;
4. if t is a data term and P a path term, then tP is a data term.

To illustrate these definitions, suppose that we are interested in the third chapter of second volume of Knuth, and suppose that we have $Knuth\_Books$ as a root of persistence. Then this information can be reached by: $Knuth\_Books\ P \cdot volumes[2]\ Q \cdot chapters[3](X)$.

This assumes that a tuple with an attribute *volumes* (the list of volumes) can be reached from the persistent root *Knuth_Books* and from a volume, one can reach a tuple (or a marked union) with an attribute *chapters* (the list of chapters). Here, for instance, $P \cdot volumes[2]\ Q \cdot chapters[3]$ is a path term and $Knuth\_Books\ P \cdot volumes[2] Q \cdot chapters[3]$ a data term. The data variable X will denote the relevant chapter.

The core calculus

The *atoms* are first formed from the data terms using equality, containment, and membership. If t, t' are data terms, then $t = t', t \in t', t \subseteq t'$ are atoms. Since we are particularly interested in the specification of paths, we introduce a second kind of atoms, namely *path predicates*. If vP is a term with v a data term and P a path term, then $\langle vP \rangle$ is an atom. The interpretation of path predicates is as follows. A ground path predicate $\langle vP \rangle$ holds if an instance of P is a concrete path

[4] Strictly speaking, we do not have integer types at this stage. We can just assume that only valuations to integers will be defined when indexing lists. This issue will disappear when we introduce typing in the next section

from the root of v. An example of path predicate is given by: $\langle Knuth\_Books\ P \cdot status(X)\rangle$ which should be interpreted as P is a path from the root of $Knuth\_Books$ to a tuple (or a marked union) with an attribute *status* whose value is denoted by X.

The *literals* are obtained from atoms using conjunction ($\wedge$), disjunction ($\vee$), negation ($\neg$) and quantification over data, path and attribute variables ($\exists, \forall$). A *query* is an expression $\{x_1, \ldots, x_n \mid \varphi\}$ were $x_1, \ldots, x_n$ are the only free variables in φ.

The semantics is then (almost) standard. The semantics of data terms will be within the set **val** of values; the semantics of path terms within **path**, and that of attribute terms within **att**.

Range-Restriction

We impose range-restrictions in the style of [3]. All variables in a formula must be range restricted. The reader will find in [3], the range-restrictions coming for instance from the use of equality, membership or containment. We insist here only on the novel aspect, the range-restriction obtained from path expressions.

Consider path predicates. The role of such predicates is twofold: (i) state the existence of paths, and (ii) range restrict the variables specified on the path. More precisely, a path variable or an attribute variable are range-restricted if they occur in a path from a root of persistence or from a range-restricted variable. For instance, in

$$\langle Knuth\_Books\ P \cdot volumes[2]\ Q \cdot chapters[J](X) \cdot A(Y)\rangle \ \wedge\ Y = \text{“Introduction”}$$

the variables P,Q,J,X,A,Y inherit their range-restriction from $Knuth\_Books$.

A subtlety is the interpretation of path variables. Consider a database of persons with spouses and the data term *Alice P name*. For path $\rightarrow$, the data term denotes Alice's name, for path $\rightarrow husband \rightarrow$, it denotes Alice's husband's name, etc. There are alternatives for interpreting such path variables:

The semantic we choose: Paths variables are interpreted by concrete paths with no two dereferencing of objects in the same class[5]. For instance, the path $\rightarrow husband \rightarrow$ will not be considered since it would involve two dereferencings of Person. This guarantees safety and indeed as we will see the resulting language can be implemented with efficient algebraic techniques. Observe also that queries going more in depth in the search can still be specified using paths of the form $P \rightarrow P'$, etc.

A more liberal semantics: One can alternatively allow paths that are not visiting twice the same object (vs. the same class). This forces to consider paths of unbounded length, i.e., length determined by the data and not the schema and to introduce a loop detection mechanism.

In hypertext applications, navigation is crucial and the liberal semantics should be used. In this paper, we use the restricted path semantics. We believe that such form of recursive navigation within the data structure is not necessary for structured documents. (Recursion can always be simulated with method calls.)

Observe that the language allows to query paths and attributes. We consider examples of queries involving path expressions:
In which attribute, can “Jo” be found?

$$\{A \mid \exists P(\langle Knuth\_Books\ P \cdot A(X)\rangle \wedge X = \text{“}Jo\text{”})\}$$

[5]This is a forward pointer since we didn't mention typing yet.

Which paths lead to "Jo"?

$$\{P \mid \langle Knuth\_Books\ P(X)\rangle \wedge X = \text{“}Jo\text{”}\}$$

What are the new paths in *Doc*?

$$\{P \mid \langle Doc\ P\rangle \wedge \neg\langle Old\_Doc\ P\rangle\}$$

What are the new titles in *Doc*?.

$$\{X \mid \exists P(\langle Doc\ P \cdot title(X)\rangle) \wedge \neg\exists P'(\langle Old\_Doc\ P' \cdot title(X)\rangle)$$

Finally it must be stressed that any O_2SQL query of the form $Doc\ PATH\_p[i].ATT\_a(x)\ldots$ can be translated into a calculus expression of the form:

$$\{[P, I, A, X, \ldots] \mid \langle Doc\ P[I] \cdot A(X)\ldots\rangle\}.$$

Interpreted Predicates and Functions

Our calculus also uses interpreted functions and predicates in the style of [3]. We assume that a set of *interpreted functions* and a set of *interpreted predicates* are given. For instance, for information retrieval, pattern matching is essential. This can be captured by appropriate interpreted functions and predicates. For instance, we assume that we have an interpreted predicate `contains` that we can use with patterns.

Interpreted predicates and functions may be useful also for the path and attribute domains. For instance, the functions `length` and `name` of the previous section can be used as follows:

$$\{X \mid \exists P(\langle Knuth\_Books\ P(X) \cdot title\rangle \wedge\ \texttt{length}(P) < 3)\}$$

$$\{X \mid \exists P, A(\langle Knuth\_Books\ P \cdot A\rangle \wedge\ \texttt{name}(A)\texttt{contains}\ \text{“}(t|T)itle\text{”}\ \wedge\ \texttt{length}(P) < 3)\}.$$

Finally, observe that the result of a query is always a set[6]. To obtain lists, interpreted functions such as `set_to_list` or `sort_by` could be introduced in the language. Consider a persistent root *MyList* of type $[(a : string + b : string)]$, i.e., a list of a or b-strings. A list of the b-strings occurring after an a-string is given by:

$$\{Y \mid Y = \texttt{set\_to\_list}(\{X \mid \exists I, J(\langle MyList[I] \cdot a\rangle \wedge\ \langle MyList[J] \cdot b(X)\rangle \wedge I < J)\})\}.$$

This query also illustrates the nesting of queries in a calculus *a la* [3].

5.3 Typing

Typing is a fundamental programming discipline. We focus here on only one of its aspects crucial in databases, namely, its role for the algebraization of programs and therefore their optimization.

For us, typing is essentially a consequence of range restriction. Given a formula φ over a schema S, one infers the precise types of each data variable basically by following the testing for range-restriction: once the range of a variable is known, it determines its type. Let us consider query:

$$\{X \mid \exists P(\langle Knuth\_Books\ P \cdot sections\{X\}\rangle) \wedge\ X \cdot title = Y \wedge Y\ \textbf{contains}\ \text{“type”}\}.$$

Then X obtains its range-restriction from *Knuth_Books* and has the type $\tau_{section}$; and Y obtains its range-restriction from X and has the type **string**. However, the polymorphism obtained

[6]This is also a limitation of standard relational calculus.

from using attribute and path variables complicates the issue. For instance, consider variable X in the formula $\exists P(\langle Knuth\_Books\ P(X) \cdot title \rangle)$.

Suppose that X may a volume, chapter, section or subsection. The type of X is a union:

$$(\alpha_1 :\ \tau_{volume} + \alpha_2 :\ \tau_{chapter} + \alpha_3 :\ \tau_{section} + \alpha_4 :\ \tau_{subsectn})$$

where α_i are system supplied attribute names. Observe that (as usual when considering union types) this may result in a combinatorial explosion and resulting types which are unions of many types. Observe also that the "interesting" valuations may also be restricted by the types as in:

$$\exists P(\langle Knuth\_Books\ P(X) \cdot title \rangle \wedge \text{ "D. Scott"} \in X \cdot review).$$

If only chapters have reviewers, then only valuations of X with chapters may occur in the result. We now have to make two comments. Firstly, strictly speaking, the type of X is a marked union, so we should be using $X \cdot \alpha_i \cdot review$ for some i (see below **Important Omissions**). Secondly, what is the meaning of $X \cdot review$ when X is of type, say $\tau_{subsectn}$. We will assume that each atom where this occurs is **false**. It is important to see that this is not a way of turning off the type checking: if no alternative of the type union has an attribute *review*, this leads to a type error.

To conclude this section, let us reconsider the example of letters with *to* and *from* fields to illustrate the subtlety of marked union and its interaction with ordered tuples in the formal setting.

Example Consider a root of persistence *Letters* which is of type

$$[(a_1 : [from : string, to : string, content : string] + a_2 : [to : string, from : string, content : string])]$$

i.e. a list of tuples where the attribute *to* comes before or after the attribute *from*. If we know the exact structure, it suffices to query:

$$\{Y \mid \exists I \langle Letters[I] \cdot a_1(Y) \rangle\}$$

to obtain the letters starting with the attribute *from*. Else, a more involved equivalent query uses the ordering of the tuple:

$$(\dagger)\{Y \mid \exists A, I, J, K(\langle Letters[I] \cdot A(Y)[J] \cdot to \rangle \wedge \ \langle Letters[I] \cdot A[K] \cdot from \rangle \ \wedge J < K)\}.$$

We are using here the fact that a tuple is also an heterogeneous list. In the formula, $Letters[I]$ denotes the I-th letter, A is an attribute (a_1 or a_2), Y is the desired letter, J is the rank of the attribute *to*, and K that of the attribute *from*.

In ($\dagger$), there is still some inelegance in the use of a variable (A) to denote an attribute (a_1 or a_2) not present in the original view of the document. To solve the problem we use the following syntactic sugaring.

Important Omissions

Marking attributes (attributes used as markers in the marked union) can be omitted in queries. In O_2SQL (see Section 4.2) this was referred as *implicit selectors*. This is clearly at the cost of some little extra work for the type checker but is imposed by the fact that the user ignores these attributes. This allows to directly "project" on the attribute *to* as in:

$$\{X \mid \exists I \langle Letters[I] \cdot to(X) \rangle\}$$

to obtain the set of letter recipients. This also allows to slightly simplify query ($\dagger$):

$$\{Y \mid \exists I, J, K(\langle Letters[I](Y)[J] \cdot to \rangle \wedge \ \langle Letters[I][K] \cdot from \rangle \wedge J < K)\}.\Box$$

5.4 Algebraization (sketch)

First consider the restriction of the calculus obtained by disallowing attribute and path variables. An algebra can be obtained in the spirit of the algebras for complex objects (e.g., [3, 12]). To handle union of types, a variant-based selection (using implicit selectors) over heterogeneous sets (or lists) has in particular to be introduced. Next, consider the subset of the calculus obtained by allowing formulas of the form

$$(\star)\ \ \exists P_1, \ldots, P_n, A_1, \ldots, A_m(\varphi)$$

where φ contains no quantification over path or attribute variables. By analysis of the query using schema information, one can find candidate valuations for the P_i and A_j. Therefore, one can transform the query into a union of queries with no attribute or path variables. This may result in introducing new variables to quantify over the elements of a set or a list. Also, we may have to introduce marking before being able to do the union.

In general, one can show that an arbitrary calculus query can be translated into a boolean combination of queries of the form $(\star)$. This provides an algebraization of the calculus.

Remark: Although not done here, it is possible to extend the equivalence between relational calculus and algebra to this extended calculus and algebra. The mapping between O_2SQL and calculus/algebra can also be demonstrated. □

To conclude, observe that this technique would not work if we choose the liberal semantics for path expressions. Indeed, if we want to compile queries to algebra expressions in this larger setting, our algebra should include some form of transitive closure/fixpoint operator.

6 Summary

In this paper a mapping from SGML documents to an OODB was defined. This required the extension of the O_2 data model [23] to union types (in the spirit of IQL [6]) and ordered tuples. We then extended the O_2 query language to deal with these new features and with the requirements of documents retrieval such as querying data with incomplete knowledge of their structure. The formal bases for these new features uses an extension of the calculus of [3]. Although motivated by structured documents, the new query facilities should be useful to a variety of other OODB applications. Among other facilities the introduction of paths as first class citizens allows users to query data (and to some extent schema) without exact knowledge of the schema in a simple and homogeneous fashion.

The work described here is currently being implemented on top of the O_2 DBMS. In particular, an extension of the Euroclid SGML parser [16] 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 appropriate semantic actions. The extension of O_2SQL is also being designed. Optimization is crucial in this context. We mentioned already on-going studies for the integration of full-text indexing facilities. Query optimization techniques were already presented in [4]. An other key aspect is that of providing the means to update the document from the database. The update semantics for this context is the topic of [5].

Acknowledgments: We are grateful to O_2Technology, Euroclid and AIS Berger-Levrault for their technical support during this project. We also thank J. Stein, B. Amann, C. Lecluse, A. Rizk and A. M .Vercoustre.

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] S. Abiteboul and C. Beeri. On the Power of Languages for the Manipulation of Complex Objects. Rapports de Recherche 846, VERSO, INRIA, BP. 105, 78153 Le Chesnay, France, 1988.

[4] S. Abiteboul, S. Cluet, and T. Milo. Querying and Updating the File. In *VLDB'93*, pages 73–84, Dublin, Ireland, August 1993.

[5] S. Abiteboul, S. Cluet, and T. Milo. More on Updating the File. Rapports de recherche, VERSO, INRIA, BP. 105, 78153 Le Chesnay, France, 1994.

[6] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *SIGMOD'89*, pages 159–173, Portland Oregon, June 1989. ACM.

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

[8] 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.

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

[10] E. Bertino, F. Rabitti, and S. Gibbs. Query Processing in a Multimedia Document System. *ACM Transactions on Office Information Systems*, 6(1):1–41, January 1988.

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

[12] S. Cluet and C. Delobel. A General Framework for the Optimization of Object-Oriented Queries. In *SIGMOD'92*, pages 383–392, San Diego, California, June 1992. ACM.

[13] M. P. Consens and A. O. Mendelzon. GraphLog: A Visual Formalism for Real Life Recursion. In *PODS'90*, pages 404–416, Nashville Tennesee, April 1990.

[14] P. Dadam and V. Linnemann. Advanced Information Management (AIM): Advanced database technology for integrated applications. *IBM Systems Journal*, 28(4):661–681, 1989.

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

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

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

[18] G. H. Gonnet and F. W. Tompa. Mind Your Grammar: a New Approach to Modeling Text. In *VLDB'87*, pages 339–346, Brighton, 1987.

[19] R. H. Güting, R. Zicari, and D. M. Choy. An Algebra for Structured Office Documents. *ACM Transactions on Office Information Systems*, 7(4):123–157, April 1989.

[20] M. Gyssens and J. Paredaens. A Grammar-Based Approach towards Unifying Hierarchical Data Models. In *SIGMOD'89*, pages 263–272, Portland Oregon, 1989. ACM.

[21] N. Ide, J. Le Maitre, and J. Véronis. Outline of a Model for Lexical Database. *Information Processing & Management*, 29(2):159–186, 1993.

[22] I. Jacobs and L. Rideau-Gallot. A CENTAUR tutorial. Rapports Techniques 140, INRIA, BP. 105, 78153 Le Chesnay, France, July 1992.

[23] P. Kanellakis, C. Lecluse, and P. Richard. Introduction to the Data Model. In F. Bancilhon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of* O_2, chapter 3, pages 61–76. Morgan Kaufmann Publishers, San Mateo, California, 1992.

[24] M. Kifer, W. Kim, and Y. Sagiv. Querying Object-Oriented Databases. In *SIGMOD'92*, pages 393–402, San Diego, California, June 1992. ACM.

[25] W. L. Lee and D. Woelk. Integration of Text Search with ORION. *Data Engineering*, 13(1):56–62, March 1990.

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

[27] R. Sacks-Davis, W. Wen, A. Kent, and K. Ramamohanarao. Complex Object Support for a Document Database System. In *Thirteenth Australian Computer Science Conference*, pages 322–333, Victoria, Australia, 1990. Monash University.

[28] J.D. Ullman. The Interface between Language Theory and Database Theory. In *Theoritical Studies in Computer Science*, pages 133–151. Academic Press, 1992.